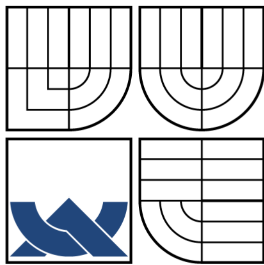
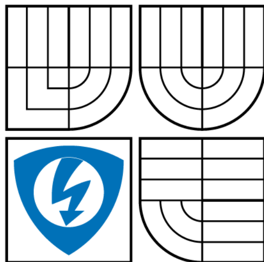


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND
COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

VYUŽITÍ PARALELIZOVANÝCH VÝPOČTŮ V PROSTŘEDÍ MATLAB VE ZPRACOVÁNÍ OBRAZU

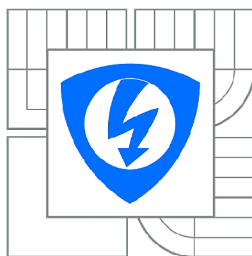
USE OF PARALLEL MATLAB PROGRAMMING FOR IMAGE PROCESSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE LUKÁŠ PRIŠŤ
AUTHOR

VEDOUCÍ PRÁCE Ing. JAN ŠPIŘÍK,
SUPERVISOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Semestrální práce

bakalářský studijní obor

Teleinformatika

Student:	Lukáš Prišť	***ID:***	128666
Ročník:	3	***Akademický rok:***	2012/2013

NÁZEV TÉMATU:

Využití paralelizovaných výpočtů v prostředí MATLAB ve zpracování obrazu

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte možnosti paralelizování výpočtů na vícejádrových procesorech ve zpracování obrazu (zejména po jednotlivých blocích) v prostředí MATLAB. Nastudované postupy pak aplikujte do funkcí, které budou rekonstruovat obraz pomocí vybrané transformace nebo nedourčených systémů lineárních rovnic.

DOPORUČENÁ LITERATURA:

[1] BRÄUNL, Thomas. Parallel image processing. Berlin: Springer, 2010, ix, 203 s. ISBN 978-3-642-08679-3.

[2] KEPNER, Jeremy. Parallel MATLAB for multicore and multinode computers. Philadelphia: Society for Industrial and Applied Mathematics, 2009, xxv, 253 s. ISBN 978-0-898716-73-3.

Termín zadání: 5.10.2012 ***Termín odevzdání:*** 12.12.2012

Vedoucí práce: Ing. Jan Špiřík

Konzultanti semestrální práce:

prof. Ing. Kamil Vrba, CSc.

Předseda oborové rady

UPOZORNĚNÍ:

Autor semestrální práce nesmí při vytváření semestrální práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Tato práce se zabývá využitím paralelizovatelných výpočtů pro zpracování obrazu v MATLABu pomocí dostupných paralelních knihoven a funkcí. Teoretická část rozebírá odlišnosti paralelního programování oproti běžnému sekvenčnímu přístupu, různé druhy paralelizace a nutnosti změny komunikace v paralelních systémech. Následně jsou tyto poznatky použity pro tvorbu paralelních funkcí, jejichž výkon je poté srovnáván s ekvivalentními sekvenčními funkcemi.

KLÍČOVÁ SLOVA

paralelizace, MATLAB, zpracování obrazu

ABSTRACT

This thesis examines the use of parallel MATLAB programing for image processing using available libraries and functions. The theoretical part compares the differences between the parallel and the usual sequential approach, different kinds of parallelization, the need for change of communication in parallel systems. This is later used for implementing parallel functions and comparing their effectivity to their sequential counterparts.

KEYWORDS

parallelization, MATLAB, image processing

PRIŠŤ, Lukáš *Využití paralelizovaných výpočtů v prostředí MATLAB ve zpracování obrazu*: bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2013. 54 s. Vedoucí práce byl Ing. Jan Špiřík,

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Využití paralelizovaných výpočtů v prostředí MATLAB ve zpracování obrazu“ 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/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno . . . .

(podpis autora)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu semestrální práce panu Ing. Janu Špiříkovi, za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno . . . .

(podpis autora)

OBSAH

Úvod **7**

1 Teoretický úvod do paralelizace **8**
1.1 Vývoj výpočetního výkonu 8
1.2 Architektury počítačů 10
1.3 Typy paralelizace 12
1.3.1 SIMD systémy 13
1.3.2 Komunikace a redukce vektorů 16
1.4 Paralelizace v MATLABu 17
1.4.1 Rozhraní paralelních funkcí 19
1.4.2 Kritéria pro použití paralelizace 20
1.4.3 Kdy použít paralelizaci 21
1.4.4 Omezení paralelních cyklů 21

2 Zpracování obrazu **22**
2.1 Převod obrazu do odstínů šedi 22
2.2 Extrapolace obrazu 22
2.2.1 Extrapolace obrazu pomocí algoritmu K–SVD 22

3 Výsledky studentské práce **25**
3.1 Implementace paralelizace krátkého kódu 25
3.1.1 Výsledky sériové implementace 25
3.1.2 Výsledky paralelní implementace 26
3.2 Implementace paralelizace na dlouhém kódu 27
3.2.1 Výsledky sériové implementace 27
3.2.2 Výsledky paralelní implementace 27

4 Závěr **28**

Literatura **29**

Seznam symbolů, veličin a zkratek **31**

Seznam příloh **32**

A Ukázky kódu **33**
A.1 Sériová implementace převodu obrázku do odstínů šedé 33
A.2 Paralelní implementace převodu obrázku do odstínů šedé 34
A.3 Extrapolate OMP 35

A.4 OMP Inpaint 39
A.5 Extrapolate Dictionary 43
A.6 Převod z pole buněk do jednorozměrného pole 53
A.7 Vytvoření pole koeficientů pro další použití 54

ÚVOD

Paralelizace je bezpochyby velmi perspektivní technologií. V dnešní době se ve velké míře uplatňuje při vědeckých výpočtech na grafických kartách (GPU) a to především díky jejich vysokému výpočetnímu výkonu. Za zmínění stojí použití grafických karet NVIDIA K20x v nejrychlejším superpočítači na světě Titan – Cray XK7 [16]. Dále může být paralelizace levným způsobem zrychlení běhu kódu bez nutnosti výměny hardwaru. Pokud v blízké budoucnosti dojde ke zpomalování růstu rychlosti procesorů, jak to předpokládají výzkumy [6], bude paralelizace jednou z cest, kterou budou společnosti využívat pro zrychlování běhu svých aplikací. Že se nebude jednat pouze o vedlejší vývojovou větev napovídá i fakt, že s využitím této technologie počítá například Microsoft a poskytuje již nyní paralelní prostředí pro vývojáře [10].

Každý kód nelze paralelizovat, proto je důležité najít správné části kódu vhodné pro paralelizaci. Nejčastěji se jedná o smyčky splňující určitá kritéria (viz 1.4.2). Cílem je najít v kódu sekce vhodné k paralelizaci, provést ji a srovnat časovou náročnost obou implementací.

1 TEORETICKÝ ÚVOD DO PARALELIZACE

1.1 Vývoj výpočetního výkonu

Spoluzakladatel společnosti Intel Gordon E. Moore v roce 1965 v rozhovoru uvedl, že počet komponentů v integrovaných obvodech se zdvojnásobuje každé dva roky od doby jejich vynálezu roku 1958. Předpokládal, že tento trend bude pokračovat alespoň následujících 10 let [11]. Jeho předpověď se ukázala jako nečekaně přesná a nyní je známá jako Moorův zákon. Exponenciální růst počtu tranzistorů v obvodu a následný růst výpočetního výkonu značně ovlivnil technologický pokrok 20. a 21. století. Tento trend dále pokračuje viz obrázek 1.1, avšak podle výroční zprávy ITRS z roku 2010 se růst zpomaluje a od roku 2013 by se měla rychlost s jakou narůstá výpočetní výkon ještě zpomalit na zdvojnásobení jednou za tři roky [6].

Obr. 1.1: Počty tranzistorů v mikroprocesorech a křivka Moorova zákona [11]

Ekonomickým důsledkem Moorova zákona je Rockův zákon, který říká, že zároveň s výpočetním výkonem rostou exponenciálně také náklady na výrobu těchto čipů. Sám Moore v roce 2005 prohlásil, že nárůst nemůže pokračovat do nekonečna a tranzistory nakonec narazí na limity miniaturizace. Předpokládejme, že zhruba za deset let přestane Moorův zákon platit. Již nyní můžeme vidět zpomalování tempa růstu výkonu při použití běžné křemíkové technologie [6]. Na příklad dnešní čipy Pentium používají vrstvu o tloušťce 20 atomů ($4,4$ nm), při jejich dalším zmenšování vrstvy narazíme na limit 5 atomů ($1,1$ nm). S tím nastanou problémy s limitem křemíku stanoveným zákonem termodynamiky a kvantové mechaniky. Jedná se o

- příliš vysokou teplotou – čip by se roztavil,
- uplatnění se kvantové teorie, Heisenbergova principu neurčitosti. Nevíme, kde se nachází elektrony.

Hledání nových materiálů a způsobů, jak nadále zvyšovat výpočetní výkon, je nezbytností. Snaha implementace 3D čipů či využití principu paralelizace s ohledem na Moorův zákon bude taktéž s postupem času nedostačující. Již nyní se setkáváme s pokusem o nasazení zajímavých technologií s využitím nových materiálů pro výrobu tranzistorů:

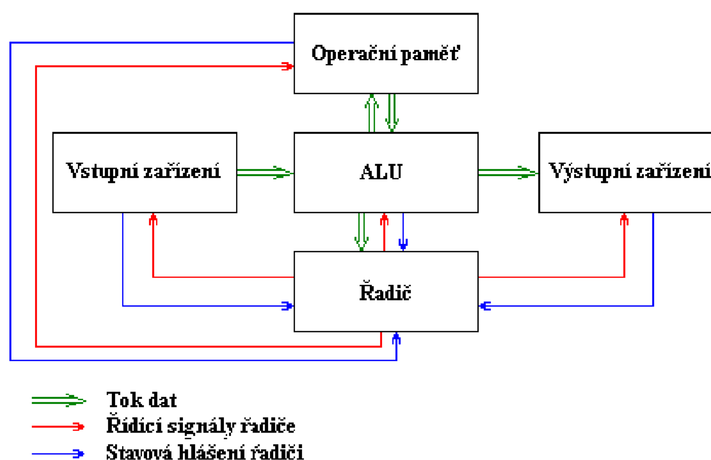
- DNA počítače,
- optické počítače,
- molekulární počítače,
- kvantové počítače.

Všechny tyto technologie však mají určité problémy. Výroba velkého množství molekulárních tranzistorů je velmi složitá a pomalá. V kvantových počítačích, kde pro výpočty používáme dva atomy, které jsou provázané a změnou stavu jednoho z nich vyvoláme změnu v druhém atomu, je velmi obtížné udržení synchronizace těchto atomů. Pro zajímavost: nejsložitější výpočet provedený na kvantovém počítači je $3 \cdot 5 = 15$. To samo o sobě nezní příliš zajímavě, tento výpočet byl ale proveden na pouhých pěti atomech.

Všechny tyto nové technologie jsou bez pochyby zajímavé, v nejbližší době však předpokládám uplatnění hlavně využití parelelizace, které se budu dále věnovat.

1.2 Architektury počítačů

Existuje několik základních představ o architektonické stavbě počítače. Většina dnešních počítačů vychází z Von Neumannovi koncepce, která vznikla kolem roku 1946. Schéma této architektury je uvedeno na následujícím obrázku 1.2.



Obr. 1.2: Von Neumannova koncepce [3]

Podle ní má počítač tyto základní části s následujícími funkcemi:

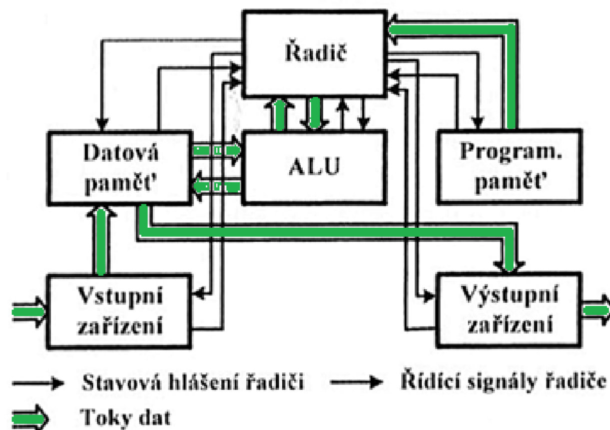
- Aritmetickologická jednotka – provádí všechny výpočty a operace.
- Řadič – řídí činnost ostatních částí počítače, se kterými komunikuje pomocí řídících signálů a od kterých dostává odpovědi ve formě stavových hlášení.
- Vstupní a výstupní zařízení – slouží pro vstup programu a dat, respektive pro zobrazení výstupu programu.
- Operační paměť – slouží k uchování běžících procesů a dat.

V této koncepci jsou tedy zpracovávaná data i programy uloženy ve stejné operační paměti. Dříve byla data od programu striktně oddělena. Existuje nebezpečí, že probíhající program začne data mylně interpretovat jako program nebo naopak. Na druhé straně nám tento přístup k paměti umožňuje, aby výstupní data jednoho programu mohla být interpretována jako program a spuštěna. Jednotlivé instrukce programu jsou v paměti řazeny za sebou a vykonávány sekvenčně.

Von Neumannova architektura je tedy ryze sekvenční a ve své čisté podobě nepředpokládá žádný paralelismus. To je na jedné straně velká výhoda, neboť sekvenční postup je přirozenější a intuitivnější než postup paralelní. Tvorba programů, které předpokládají čistě sekvenční zpracování, je pak snazší než tvorba paralelních. I to je zřejmě jeden z důvodů, proč se von Neumannova koncepce udržela až dodnes, zatímco alternativní koncepce, podporující větší míru paralelismu, nejsou zdaleka

tak úspěšné. Navíc sekvenční vykonávání instrukcí znamená, že v jednom okamžiku bude pracovat pouze daná část počítače a zbytek bude neaktivní [13].

Dalším modelem je Harwardská architektura počítače 1.3.



Obr. 1.3: Harwardská koncepce [3]

Název pochází z počítače postaveného na této architektuře Harward Mark I. a od von Neumannovi architektury se liší především:

- rozdělením paměti programu a paměti dat;
- sběrnicemi oddělenými pro jednotlivé směry;
- nepřímým připojením vstupní a výstupní jednotky k ALU.

Je zřejmé, že nemusíme současně použít stejný typ paměti pro data a pro program. Můžeme například pro program použít paměť `ROM` a pro data paměť typu `RAM`. Rozdělení paměti programu a dat nám umožňuje paralelní přístup k oběma pamětem. Použitím paměti `ROM` pro uložení programu můžeme navíc zvýšit bezpečnost systému. Využitím tohoto paralelního přístupu k pamětem můžeme velmi rychle zpracovat velké objemy dat.

Dnešní rychlé procesory spojují obě architektury tak, že uvnitř procesoru je použita Harwardská architektura, která svoji vyrovnávací paměť dělí na paměť instrukcí a paměť dat. Z vnějšku se však celý procesor chová jako procesor s von Neumannovou architekturou, protože data načítá z operační paměti najednou.

1.3 Typy paralelizace

Existují tři základní modely pro paralelní zpracování dat:

- pipeline processing,
- asynchronní paralelní zpracování,
- synchronní neboli data–paralelní zpracování.

Počítačové systémy používající asynchronní paralelní zpracování dat se nazývají MIMD systémy (*multiple instructions, multiple data*). V těchto systémech má každý procesor PE (*processing element*) svoji vlastní správu řízení přístupu a v podstatě vykonává svůj vlastní program nebo část programu.

Synchronní paralelní počítače jsou označovány zkratkou SIMD (*single instruction, multiple data*) nebo také data–paralelní systémy. V těchto systémech jsou všechny procesory nebo–li PE řízeny pomocí jednoho kontrolního procesoru. Všechny PE vykonávají stejné příkazy současně na svých lokálních datech nebo jsou neaktivní. Používají sekvenční řízení přístupu a neběží zde žádné asynchronní procesy. Tento model je snáze programovatelný, jelikož jsou všechny procesory víceméně synchronizovány po každém kroku. Není tudíž potřeba implementovat rozsáhlé synchronizační algoritmy, které jsou náchylné k chybám. Architektura SIMD procesorů je jednodušší ve srovnání s MIMD. Tyto procesory zabírají méně místa a můžou být umístěny hustěji. Jako takové mohou být SIMD systémy sestaveny z mnohem většího počtu méně výkonných PE než MIMD systémy.

Pojem masivní paralelismus, který se vztahuje k počtu procesorů daného paralelního počítače, můžeme použít v případě, pokud systém obsahuje tisíc nebo více procesorů. V současnosti dosahujeme takového stupně paralelizace pouze v SIMD systémech, neboť vyžadují použití nových programovacích technik a jiných algoritmů než běžné asynchronní paralelní zpracování.

Masivně paralelizovaný přístup přináší značné výhody zvláště pro zpracování obrazu. Protože nyní můžeme při dostatečném počtu procesorů každému pixelu přiřadit jeden procesor. Pokud je však počet pixelů větší než počet dostupných PE, použijeme virtuální PE, jejichž princip je totožný s principem virtuální paměti.

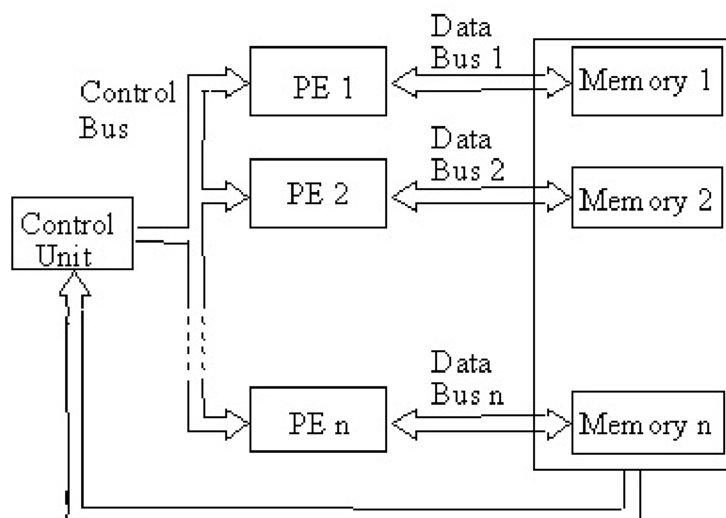
Tento přístup otevírá nové možnosti pro zpracování obrazu a v mnoha případech zjednodušuje i jejich algoritmy. Každý PE nyní vykonává nad daty svého pixelu stejnou operaci paralelně s ostatními PE, tudíž synchronizace těchto procesů ze strany programátora již není nutná.

Řešení téhož problému v asynchronních paralelních systémech je mnohem složitější. Obraz musí být rozdělen na části a každá část zpracována individuálními asynchronními procesory, přičemž každý procesor zpracuje data z každého pixelu ve své části. Problém nastává na okrajích částí obrazu přidělených jednotlivým proceso-

rům, kdy jsou pro zpracování zapotřebí data ze sousední části, která má však lokálně uložen jiný procesor. Tento problém lze překonat jednoduše tím, že obraz bude rozdělen na části, které se částečně překrývají, nebo můžeme procesory složitě synchronizovat a následně mezi nimi provést výměnu dat. Tato synchronizace je vždy nutná po dokončení výpočtu na všech procesorech, kde musí být lokální data jednotlivých procesorů zkombinována dohromady pro vytvoření nového obrazu. V asynchronních paralelních systémech je výměna dat časově náročnější, řádově tisíckrát oproti provedení jedné aritmetické operace. Hlavní snahou by tedy mělo být její co nejmenší využití. Naopak je tomu v synchronních systémech, kde operace výměny trvá stejně dlouho jako jedna aritmetická operace. Zde ji tedy můžeme využívat libovolně [2].

1.3.1 SIMD systémy

Data–paralelní systémy odpovídají synchronnímu modelu paralelizace. Řídící procesor v tomto modelu je standardní sekvenční počítač (SISD: *single instruction, single data*), ke kterému jsou připojena ostatní zařízení. Jednotlivé PE nevykovávají vlastní program ale pouze příkazy, které dostávají od řídícího procesoru. Jednotlivé PE nemají vlastní překladač příkazů, jedná se o neúplné procesory, ALU (*arithmetic logic units*) aritmeticko–logické jednotky s vlastní lokální pamětí a komunikačním kanálem. S jednoduchostí tohoto modelu souvisí také jeho omezení. Všechny PE buď ve



Obr. 1.4: Architektura SIMD

stejný okamžik vykonávají stejný příkaz, který obdrželi od řídícího procesoru nebo jsou neaktivní. Vezměme si pro příklad paralelní implementaci příkazu `IF`.

Kód 1.1: Implementace příkazu IF.

```
1  if (podmínka)   // Nyní se PE rozdělí do dvou skupin
2  {
3       KÓD
4       .    // Tento kód vykonávají pouze PE, kde je podmínka splněna,
5       .    // ostatní PE jsou neaktivní
6       .
7  }
8  else
9  {
10      KÓD
11      .    // Tato část kódu je vykonána na všech PE, které byly ...
            neaktivní
12      .    // Naopak PE, kde byla podmínka splněna, jsou nyní neaktivní
13      .
14 }
```

Jednotlivé PE jsou propojeny pomocí sítě, která jim dovoluje si mezi skupinami předávat data. Výměna dat nenastává mezi dvěma konkrétními PE, protože by muselo dojít k jejich synchronizaci, ale probíhá hromadně mezi všemi PE nebo všemi PE v určité skupině. Zatímco v asynchronních paralelních systémech je komunikace problémem, v synchronních systémech je snadná. Řídící procesor si také může vyměňovat data selektivně s jednotlivými PE nebo se všemi PE jako všesměrové vysílání.

Komunikační síť propojující jednotlivé PE by měla mít dostatečnou šířku pásma pro paralelní výměnu dat mezi PE, kde čas pro výměnu dat odpovídá času jedné aritmetické operace. Komunikace mezi PE a řídícím procesorem může představovat problém, z důvodu sekvenčního provedení. Obvykle jsou obrazová data načtena přes řídící procesor a distribuována jednotlivým PE. Sběrnice mezi nimi by měla být dostatečně rychlá. Použitím jiného způsobu distribuce dat však můžeme celý proces zrychlit. Řídící procesor může například místo odesílání dat každému PE jednotlivě, poslat pouze všem procesorům v první řadě vždy celý řádek obrazu. Odsud jsou data přeposlána jednotlivým PE v řadách pod sebou prostřednictvím rychlé paralelní sítě. Tím se sice nezmění objem dat odesílaných řídícím procesorem, ale počet PE, které musí přímo adresovat, je mnohem menší.

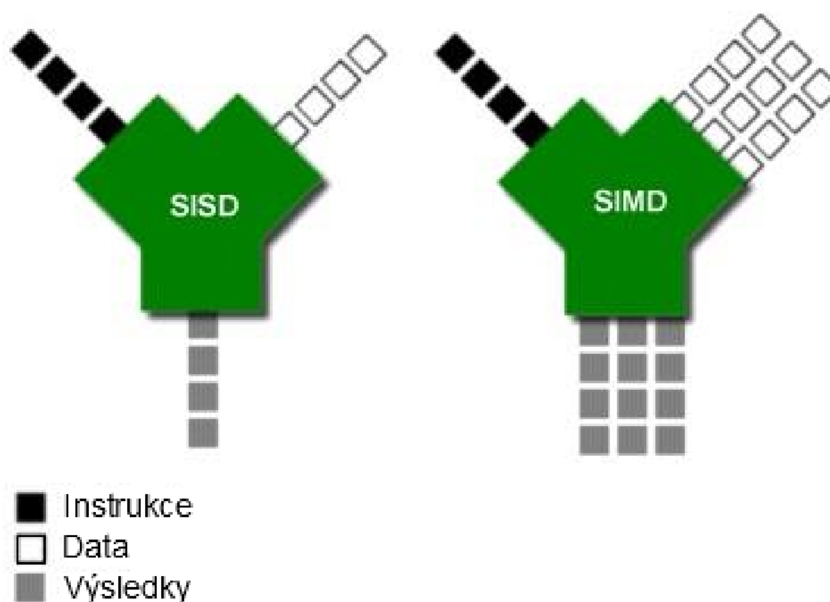
Přestože masivně paralelní systémy obsahují obrovské množství PE, může nastat situace, kdy ani toto množství nestačí. Pokud bychom chtěli například aby při práci s obrazem o rozměrech 1024x1024 pixelů měl každý pixel přiřazen jeden PE, je již nutné použít virtuální PE. Umístění virtuálních PE na fyzické PE by sice mohl mít

na starost programátor aplikace, ale je žádoucí, aby tuto často používanou funkci provádělo buď přímo samotné programovací prostředí nebo paralelní hardware.

Pokud tedy počet PE, které program vyžaduje, překročí počet reálných PE, měly by být virtuální PE programátorovi aplikace jednoduše k dispozici. Tvoří tak určitý stupeň abstrakce a mohou být implementovány v iteracích. Systém poté namapuje virtuální PE na fyzické PE, buď hardwarově nebo softwarově. Princip virtuálních procesorů se stává analogickým k principu virtuální paměti. Naopak pokud program potřebuje méně PE než daný systém obsahuje, musí nevyužité PE zůstat neaktivní a nemohou být použity pro jiný účel, jedná se limitaci data–paralelního modelu.

Vzhledem k množství procesorů používaných v data–paralelních systémech musíme při jejich programování opustit uvažování ve stylu von Neumannova modelu. Velké množství problému, zvláště těch souvisejících se zpracováním obrazu lze pomocí data–paralelních systémů řešit jednodušeji a rychleji než pomocí klasických sekvenčních programovacích jazyků, jak je ilustrováno na obrázku 1.5.

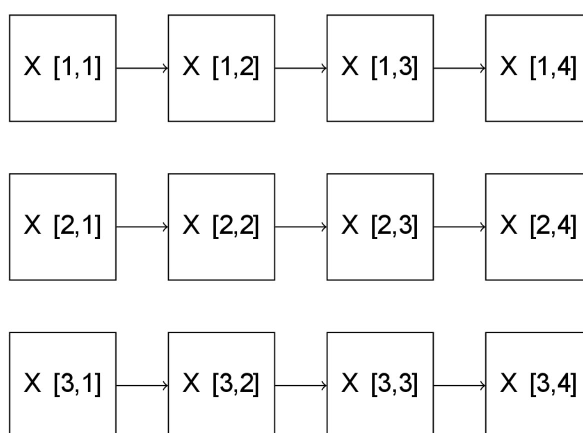
Typickým příkladem využití SIMD systému může být vytvoření negativu daného RGB obrázku. Na klasickém počítači bychom museli postupně v jednotlivých iteracích provádět na jednotlivých pixelech stejnou operaci, pokud však využijeme vlastností data paralelních systémů, celý proces se značně zrychlí.



Obr. 1.5: Srovnání architektur SISD a SIMD [15]

1.3.2 Komunikace a redukce vektorů

Výměna dat v data–paralelním systému zahrnuje vždy buď všechny PE nebo určitou podskupinu a je mnohem méně náročná než v asynchronních paralelních systémech. Synchronizovaná spojení jsou zde tvořena rychleji a síť, která je spojuje, má obvykle větší propustnost. Mnoho data–paralelních systémů obsahuje kromě připojení k obecnému typu sítě také připojení k velmi rychlým sítím typu grid. Tyto sítě slouží většinou k připojení systémů s jednoduchou strukturou. Změny ve stanovené struktuře u tohoto připojení způsobují značné poklesy rychlosti, protože výměna dat potřebuje více kroků, nebo v horším případě musí využít pomalejší obecné připojení.

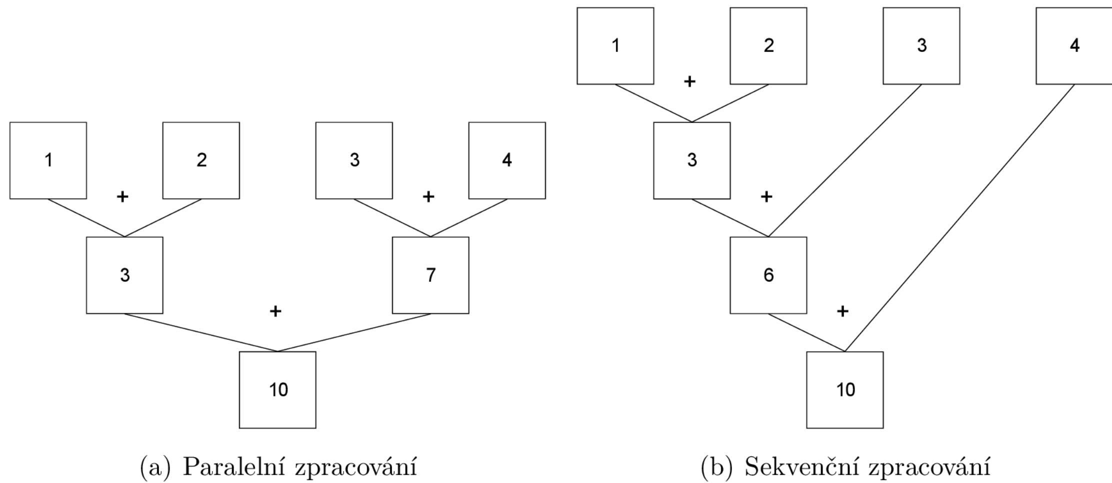


Obr. 1.6: Data–paralelní výměna dat [2]

Klíčová je pro paralelní procesory operace redukce vektorů, která může být implementována hardwarově nebo softwarově jako systémová funkce. V těchto systémech popisují vektory všechny komponenty dané proměnné nebo konstanty rozprostřené na všech PE. Vektor se tvoří tak, že všechny PE postupně pošlou svoji část proměnné viz obrázek 1.6. Pomocí redukce vektorů se poté takto vygenerovaný vektor zredukuje na skalární hodnotu viz obrázek 1.7. Používá k tomu operace sčítání, násobení, maximum, minimum, logickou operaci `AND`, `OR` atd. Zvolená operace musí vždy být asociativní a komutativní, aby výsledek nezávisel na pořadí provedení operací.

Tato vektorová redukce byla provedena tak, že všechny komponenty byly sčítány dokud nezbyla pouze jedna skalární hodnota. Zde opět vidíme, že paralelní provedení bylo mnohem efektivnější než sekvenční. Sekvenční sčítání našich 4 prvků proběhlo ve 3 krocích, zatím co paralelně tento proces potřeboval pouze 2 kroky.

Obecně tedy sekvenční sčítání n hodnot trvá $n - 1$ časových kroků. Paralelní zpracování stejného počtu prvků trvá $\log_2 n$ kroků [2].



(a) Paralelní zpracování (b) Sekvenční zpracování

Obr. 1.7: Vektorová redukce [2]

1.4 Paralelizace v MATLABu

MATLAB je v současnosti dominantním programovacím jazykem vyšší úrovně v oblasti technických výpočtů. Počet uživatelů se celosvětově pohybuje kolem jednoho milionu. MATLAB je ideálním prostředím pro zkoumání možností paralelizace různých algoritmů, protože poskytuje velké množství paralelních knihoven a nemusíme se tedy starat o detaily její implementace. Nejpoužívanější knihovnou bude `pMatlab` [8] vyvinutá na MIT Lincol Laboratory, `Parallel Computing Toolbox` [12] vyvinutý společností MathWorks, Inc. a `StarP` [14] vyvinutá na MIT, UCSB a Interactive Supercomputing, Inc. Všechny tyto knihovny poskytují přímou podporu paralelního programování pomocí distribuovaných polí a dalších modelů paralelního programování. [7]

Popis paralelních algoritmů vyžaduje zvláštní značení. Počet instancí běžícího programu je dán proměnou N_P. Ve výpočetním modelu SPMD, kde každá instance programu vykonává stejný kód, je nezbytné tyto instance dále rozlišit, proto je každému procesu přiděleno unikátní identifikační číslo značené jako P_{ID} z rozsahu $0 \ldots N_P - 1$. N_P značí kolik P_{ID} bude vytvořeno bez ohledu na počet fyzických uzlů či procesorů. Jelikož procesy mohou být spuštěny na různých výpočetních systémech zavádíme značení pro rozlišení dvou nejběžnějších implementací.

- Jako N_P*1 značíme systémy, kde N_P instancí běží na paralelním systému obsahujícím N_P uzlů, přičemž na každém uzlu běží jedno P_{ID}.
- Jako 1*N_P značíme systémy, kde všechny P_{ID} běží na jednom uzlu.

Pokud je daný uzel v systému 1*N_P pouze jedno–jádrový procesor, je nepravděpodobné, že by došlo pomocí paralelizace ke zrychlení programu. Pokud je uzel

více–jádrový procesor, je zde jistá možnost zrychlení paralelizací. Systém $1^*\mathrm{N_P}$ se může jevit jako nepříliš užitečný pro paralelní aplikace, je však velice užitečný pro testování a odlaďování chyb v paralelním programu.

Paralelní programování hojně využívá distribuovaných polí a to jak v systémech s více uzly, tak v systémech s jedním více–jádrovým uzlem. Elementy distribuovaných polí je nutné namapovat na skupinu $\mathrm{P_{ID}}$, přičemž jsou každému $\mathrm{P_{ID}}$ přiřazeny prostředky, které může využít. Matici mapujeme tak, že každé $\mathrm{P_{ID}}$ bude mít k dispozici jednu řadu.

$$\mathbf{A} : \mathbb{R}^{P(N)\times N}$$

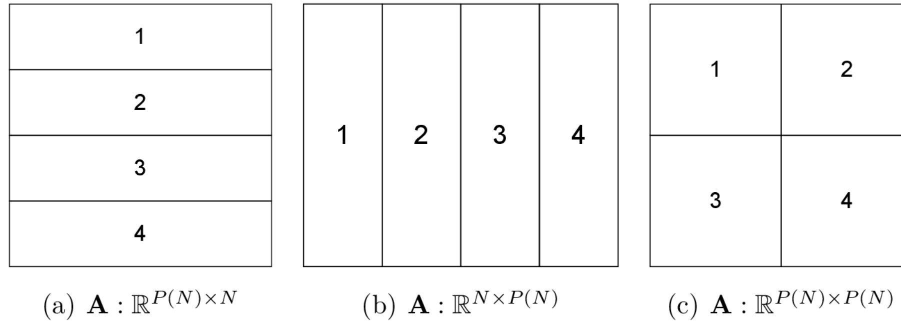
Obdobně lze namapovat matici tak, že každé $\mathrm{P_{ID}}$ bude mít k dispozici jeden sloupec.

$$\mathbf{A} : \mathbb{R}^{N\times P(N)}$$

Rozdělení matice na bloky o stejném počtu sloupců a řádku pro každé $\mathrm{P_{ID}}$ by vypadalo následovně.

$$\mathbf{A} : \mathbb{R}^{P(N)\times P(N)} \text{ nebo } \mathbf{A} : \mathbb{R}^{P(N\times N)}$$

Při použití rozdělení na bloky je třeba ještě upřesnit skupinu použitých $\mathrm{P_{ID}}$ [7].



(a) $\mathbf{A} : \mathbb{R}^{P(N)\times N}$ (b) $\mathbf{A} : \mathbb{R}^{N\times P(N)}$ (c) $\mathbf{A} : \mathbb{R}^{P(N)\times P(N)}$

Obr. 1.8: Různé způsoby mapování distribuovaných polí

Pro přístup ke konkrétnímu prvku můžeme použít standardní zápis. Pokud tedy máme distribuované pole $\mathbf{A} : \mathbb{R}^{P(N)\times N}$, tak zvolení $\mathbf{A}(i, j)$ způsobí, že $\mathrm{P_{ID}}$ které vlastní i, j–tý prvek $\mathbf{A}$ pošle jeho hodnotu všem ostatním $\mathrm{P_{ID}}$.

Můžeme také přistupovat k lokálním proměnným jednotlivých $\mathrm{P_{ID}}$ a to pomocí tečkové konvence *.loc*. Lokální část $\mathbf{A} : \mathbb{R}^{P(N)\times N}$ získáme pomocí příkazu $\mathbf{A}.loc : \mathbb{R}^{(N/N_P)\times N}$. Uvedený zápis je užitečný pokud potřebujeme přistoupit k určitým lokálním datům jednotlivých $\mathrm{P_{ID}}$, navíc tento přístup nevyžaduje další komunikaci [7].

1.4.1 Rozhraní paralelních funkcí

K implementacím dříve zmíněných paralelních postupů potřebujeme odpovídající paralelní funkce. V MATLABu je zabudována řada funkcí, které můžeme využít [9]. Pro příklad uvádím některé základní funkce.

`matlabpool` – Otevře nebo zavře skupinu zpracovatelských procesů.
Použití příkazu: `matlabpool local 4`
`parfor` – Vykoná kód následující smyčky paralelně.
Použití příkazu: `parfor i=1:25 ... end`
`spmd` – Provede následující kód paralelně ve skupině zpracovatelských procesů.
Použití příkazu: `spmd ... end`

Zde jsou ukázky a uplatnění nejdůležitějších paralelních funkcí přídavné knihovny `pMatlab`, používané v knize Jeremy Kepnera [7]. Parametry paralelního prostředí nám poskytují informace o stavu paralelního prostředí, ve kterém máme spuštěn program.

`Np` – funkce vrací celkový počet instancí MATLABu, na kterých je aktuálně spuštěn daný program. Použití příkazu: `Np;`

`Pid` – funkce vrací identitu všech instancí MATLABu, ve kterých momentálně probíhá daný program. První instance má hodnotu 0 a poslední vždy `Np-1`. Použití příkazu: `Pid;`

K tvorbě distribuovaných polí využíváme funkce, které umožňují jejich tvorbu a snadnou komunikaci mezi jednotlivými instancemi MATLABu.
`map([Np 1],{},0:Np-1)` – Vytvoří jednorozměrnou mapu, kde se na první rozměr namapuje pole.
Použití příkazu: `Amap = map([Np 1],{},0:Np-1);`
`[Np 1]` značí pole procesorů, `{}` je způsob distribuce – v tomto případě automatický, nakonec `Np-1` je seznam procesorů
`map([1 Np],{},0:Np-1)` – Vytvoří jednorozměrnou mapu, kde se na druhý rozměr namapuje pole.
Použití příkazu: `Amap = map([1 Np],{},0:Np-1);`
`zeros([N1,...,map)` – Přetížená funkce `zeros` vytvoří distribuované pole s rozdělením specifikovaném v parametru `map`.
Použití příkazu: `A = zeros(200,100,Amap);`
`local(A)` – Vrací lokální část distribuovaného pole jako běžné pole čísel.
Použití příkazu: `Aloc = local(A);`
`global_ind(A,dim)` – Vrací seznam globálních indexů, které jsou pro dané `Pid` lokální v daném rozměru `dim`.

Použití příkazu: `myI = global_ind(A,1);`
`put_local(A,Aloc)` – Zkopíruje běžné číselné pole do lokální části distribuovaného pole.
Použití příkazu: `A = pul_local(A,Aloc);`

Potřebujeme také funkce, které umožňují přemapovat distribuované pole. Tyto funkce se velice hojně používají během komunikace mezi P_{ID}.
`=` – Přiřadí data jednoho distribuovaného pole jinému distribuovanému poli.
Použití příkazu: `B(:,:)=A;`
`agg(A)` – Zkopíruje celý obsah distribuovaného pole do běžného pole čísel na hlavním P_{ID} (tedy na `Pid==0`).
Použití příkazu: `Aagg=agg(A);`

Klíčová je bodová komunikace mezi jednotlivými instancemi MATLABu. Komunikační funkce využíváme pouze pokud je to opravdu nezbytné, protože při jejich použití je ve výsledném programu obtížné odladit chyby.
`SendMsg(dest,tag,var1,var2,...)` – odešle proměnné určené parametry `var1, var2, ...` do instance MATLABu, kde `Pid==dest`. Zpráva je označena pomocí proměnné `tag`, což je obvykle číslo od 1 do 256. Použití příkazu: `SendMsg(2,9,pi,i);`
`RecvMsg(source,tag)` – přijme proměnné odeslané instancí MATLABu s `Pid==source` a odpovídajícím označením `tag`.
Použití příkazu: `[pi i]=RecvMsg(1,9);`

1.4.2 Kritéria pro použití paralelizace

Nejčastěji najde paralelizace uplatnění při tvorbě programu, který spouštíme opakovaně a pouze měníme jeho vstupní parametry. Programy se označují „trapně paralelní“ (embarasingly parallel) protože mohou být spuštěny samostatně a nepotřebují komunikovat s uživatelem. Při výběru konkrétní paralelní implementace daného programu bývá trapně paralelní přístup většinou nejvhodnější. Nejdůležitější vlastnosti, které musí kód splňovat aby mohl být paralelizován jsou:

- časté opakování výpočtu lišící se pouze parametrem,
- výsledek tohoto opakovaného výpočtu nezávisí na ostatních výpočtech.

První vlastnost znamená, že při řešení se často vyskytují výpočty, které lze provádět souběžně. Druhá vlastnost značí, že tyto souběžné výpočty používají vysoce lokalizovaná data, která jsou lokální pro jednotlivé P_{ID}. Najít výpočty, které mohou běžet souběžně není těžké. Naopak je tomu u lokalizovaných dat, která jsou klíčová pro dobrý výkon paralelního programu. Právě tyto dvě základní vlastnosti, které spojují většinu výpočtů, jsou základním požadavkem na trapně paralelní programy.

Nejdříve je třeba se rozhodnout, jak přiřadíme jednotlivé parametry různým P_{ID}. Tomuto procesu přiřazování se říká mapování. Pokud bychom tedy měli k dispozici například 4 P_{ID}, použili bychom některé z rozdělení v obrázku 1.8.

Dále je nutné zvolit, jakým způsobem získá P_{ID} data potřebná k výpočtu. Většinou se používá distribuovaných polí. P_{ID} z nich získá svá data a provede výpočet, který si uloží do své lokální proměnné. Nakonec řídící P_{ID} získá od ostatních P_{ID} jejich lokální proměnné a z nich získá konečný výsledek[7].

1.4.3 Kdy použít paralelizaci

Existují pouze dva důvody, proč využít paralelizace výpočtů:

- Časové – výpočet na jednom procesoru trvá příliš dlouho.
- Paměťové – výpočet vyžaduje více paměti, než dokáže poskytnout jeden procesor.

Nejčastěji řešíme paralelizací oba problémy současně. Časové důvody jsou na rozdíl od paměťových subjektivní. Při vyčerpání dostupné paměti mohou nastat dvě situace. Buď je vyvoláno chybové hlášení, nebo v důsledku snahy operačního systému o navýšení dostupné paměti pomocí swapování na pevný disk nastane výrazné zpomalení.

1.4.4 Omezení paralelních cyklů

Použití paralelních cyklů s sebou přináší některá omezení. Jelikož cykly probíhají současně v nedeterminovaném pořadí, nelze se spoléhat na výsledky předchozích iterací.

Hodnoty proměnných z vnějšku cyklu jsou při vstupu do cyklu smazány a je nutné vytvořit na začátku cyklu dočasnou proměnnou a nakopírovat do ní obsah „venkovní“ proměnné.

Výsledky výpočtů prováděných v jednotlivých iteracích ukládáme nejčastěji do polí, které je nutné předem naalokovat, nebo do pole buněk, pokud ukládáme v iteracích celá pole. V těchto polích můžeme výsledky indexovat pouze pomocí čísla právě probíhající iterace.

2 ZPRACOVÁNÍ OBRAZU

2.1 Převod obrazu do odstínů šedi

Převod obrazu do odstínů šedi patří mezi jednoduché operace. Pro převod je použita jasová rovnice (2.1), která odstraní barevné složky původního obrazu a zachová pouze informaci o jasové intenzitě.

$$Y' = 0,2989 \cdot R + 0,5870 \cdot G + 0,1141 \cdot B. \tag{2.1}$$

Tento převod se používá především v aplikacích, kde potřebujeme pro identifikaci šedotónovou složku, například detekce hran v obraze, infrakamery, televizní vysílání, atd.

2.2 Extrapolace obrazu

Extrapolace obrazu patří mezi nepříliš běžně používané metody ve zpracování obrazu, dalo by se říct, že se jedná o zvláštní případ dokreslování obrazu. Zásadní rozdíl spočívá v tom, že při dokreslování obrazu předpokládáme uzavřenou topologii, což při extrapolaci není splněno.

2.2.1 Extrapolace obrazu pomocí algoritmu K–SVD

Algoritmus K–SVD slouží k adaptivnímu nalezení slovníku pro řídkou interpretaci signálu. Využívá k tomu nedourčených soustav lineárních rovnic a jejich řídkých řešení [4]. Řídké řešení je takové, které obsahuje co nejvíce nulových prvků. Hledání řídkého řešení problému P_0 je definováno následovně:

$$P_0 : \min_{\mathrm{x}} \|\mathrm{x}\|_0 \quad \text{vzhledem k} \quad \mathbf{y} = \mathbf{D}\mathbf{x}, \tag{2.2}$$

kde $\mathbf{y}$ je známý signál (obraz), který chceme rekonstruovat, $\mathbf{D}$ je slovník, který se skládá z K atomů (základních signálů) ve sloupcích a m řádků, které určují délku atomů. Všechna $\mathbf{x}$, která splňují $\mathbf{D}\mathbf{x} = \mathbf{y}$, nazýváme přípustná řešení.
Optimalizovaný problém je definován v ℓ_0. Pokud platí $\|x\|_0 \ll K$ pro $\mathbf{x} \in \mathbb{R}^n$, můžeme říct, že $\mathbf{x}$ je řídké.
Hledání řídkého řešení v ℓ_0 je NP-těžký problém. Vzhledem k velikostem slovníků a signálů je hledání řešení v reálném čase nemožné. K jeho určení se používají stíhací algoritmy. To umožňuje předefinovat řešení problému jako:

$$P_0 : \min_{\mathrm{x}} \|\mathrm{x}\|_0 \quad \text{vzhledem k} \quad \|\mathbf{y} = \mathbf{D}\mathbf{x}\|_2 \leq \epsilon, \tag{2.3}$$

kde ϵ je chyba řešení.

Ve většině aplikací se uvažuje neměnný slovník $\mathbf{D}$ [1],[17]. Algoritmus K–SVD slouží k adaptivnímu sestavení slovníku, který signál řídce reprezentuje. Název K–SVD se skládá ze zkratky algoritmu SVD (Singular Value Decomposition), který je proveden K–krát, kde K je počet sloupců ve slovníku $\mathbf{D}$. Algoritmus K–SVD je v podstatě zobecnění algoritmu K–means. Skládá se ze dvou základních kroků:

- nalezení řídkých řešení,
- aktualizace slovníku.

Nejprve se slovník inicializuje a všechny sloupce (atomy) musí být ℓ_2–normalizované. Jako výchozí slovník můžeme použít náhodnou matici nebo DCT slovník. Problém, který má algoritmus K–SVD řešit, můžeme formulovat jako:

$$\min_{\mathbf{D},\mathbf{X}} \{\|\mathbf{Y} - \mathbf{D}\mathbf{X}\|_F^2\} \quad \text{vzhledem k} \quad \forall i, \|\mathbf{x}_i\|_0 \leq T_0, \tag{2.4}$$

kde $\mathbf{Y}$ je matice obsahující trénovací vzorky $\{\mathbf{y}_i\}_{i=1}^N$ ve sloupcích, $\mathbf{X}$ je matice odpovídajících koeficientů, T_0 je povolená odchylka reprezentace a F označuje Frobeniovu normu, která je definována jako:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}. \tag{2.5}$$

V prvním kroku předpokládáme, že $\mathbf{D}$ je pevně dané. Penalizační člen můžeme vyjádřit jako:

$$\|\mathbf{Y} - \mathbf{D}\mathbf{X}\|_F^2 = \sum_{i=1}^{N} \|\mathbf{y}_i - \mathbf{D}\mathbf{x}_i\|_2^2. \tag{2.6}$$

Nyní můžeme rozdělit 2.4 do N problémů:

$$i = 1, 2, \ldots, N \quad \min_{\mathbf{x}_i} \{\|\mathbf{y}_i - \mathbf{A}\mathbf{x}_i\|_2^2\} \quad \text{vzhledem k} \quad \|\mathbf{x}_i\|_0 \leq T_0. \tag{2.7}$$

Tento problém lze pak již řešit pomocí známých algoritmů pro hledání řídkého řešení nedourčených soustav lineárních rovnic, například OMP.

V druhém kroku aktualizujeme slovník $\mathbf{D}$. Předpokládáme, že $\mathbf{A}$ a $\mathbf{X}$ jsou neměnné. V každém kroku bude aktualizován jen daný atom (sloupec) $\mathbf{d}_k$ a jeho koeficienty $\mathbf{x}_T^k$ v matici $\mathbf{X}$. Díky tomu můžeme definovat množiny ω_k, které se budou skládat z indexů vektorů $\{\mathbf{y}_i\}$, které používají atom $\mathbf{d}_k$, tedy kde $\mathbf{x}_T^k$ je nenulové:

$$\omega_k = \{i |\ 1 \leq i \leq N,\ \mathbf{x}_T^k(i) \neq 0\}. \tag{2.8}$$

Poté definuje matici chyb $\mathbf{E}_k$, která vyjadřuje chybu pro všech N vzorků při chybějícím k–tém atomu:

$$\mathbf{E}_k = \mathbf{Y} - \sum_{j \neq k} \mathbf{d}_j \mathbf{x}_T^j. \tag{2.9}$$

S pomocí těchto podmínek můžeme přepsat 2.4 jako:

$$\|\mathbf{Y}-\mathbf{D}\mathbf{X}\|_F^2 = \left\|\mathbf{Y}-\sum_{j=1}^{K}\mathbf{d}_j\mathbf{x}_T^j\right\|_F^2 = \left\|\left(\mathbf{Y}-\sum_{j=1}^{K}\mathbf{d}_j\mathbf{x}_T^j\right)-\mathbf{d}_k\mathbf{x}_T^k\right\|_F^2 = \left\|\mathbf{E}_k-\mathbf{d}_k\mathbf{x}_T^k\right\|_F^2. \tag{2.10}$$

Pro zjednodušení aplikujeme množiny ω_k na matice $\mathbf{E}_k$ tak, že vybereme pouze sloupce odpovídající ω_k, čímž získáme $\mathbf{E}_k^R$. Použitím SVD algoritmu rozložíme $\mathbf{E}_k^R$ na:

$$\mathbf{E}_k^R = \mathbf{U}\Delta\mathbf{V}^T. \tag{2.11}$$

Posledním krokem při aktualizaci slovníku je nahrazení aktuálního atomu (sloupce) slovníku $\mathbf{d}_k$ prvním sloupcem matice $\mathbf{U}$. A současně odpovídající vektor koeficientů $\mathbf{x}_R^k$ nahradíme prvním sloupcem z matice $\mathbf{V}$ vynásobeným $\Delta(1,1)$.

Algoritmus K–SVD používáme pro trénování slovníku ze známé části obrazu. Extrapolace obrazu probíhá ve smyčce. V každém kroku extrapolujeme obraz pouze pro 1 pixel a současně počítáme pouze jednu řadu nebo sloupec, který je poté přidán do obrazu. Předpokládáme, že extrapolovaná část je bez chyb. V následující iteraci používáme extrapolovaný obraz z předchozí iterace. V každé smyčce překrýváme všechny bloky. Extrapolace je silně závislá na velikosti bloku naučeného slovníku. [17]

3 VÝSLEDKY STUDENTSKÉ PRÁCE

Měření proběhlo na počítači s konfigurací:

- AMD Phenom(tm) X6 1090T 3.20GHz,
- 4GB RAM,
- Windows 8 Pro 64-bit,
- MATLAB R2010a.

Pro paralelní zpracování jsem použil skupinu 6 zpracovatelských úloh.

3.1 Implementace paralelizace krátkého kódu

Jako jednoduchý příklad využití paralelizace jsem implementoval algoritmus převodu barevného obrázku na odstíny šedé s použitím jasové rovnice (2.1). Pro porovnání je algoritmus implementován nejenom paralelně (A.2), ale i sériově (A.1).

Porovnání obou metod bylo provedeno deseti měřeními času potřebného pro výpočet na obrázku o rozměrech 512x512 pixelů. Předpokládám, že paralelní implementace algoritmu bude potřebovat na převod obrázku méně času.

3.1.1 Výsledky sériové implementace

Výsledky při použití sériové implementace kódu (A.1) jsou uvedeny v následující tabulce 3.1.

Tab. 3.1: Doba trvání sériové implementace

Pořadí měření	[-]	1	2	3	4	5
Naměřený čas	[s]	0,0830	0,0994	0,0821	0,0818	0,0829
Pořadí měření	[-]	6	7	8	9	10
Naměřený čas	[s]	0,0820	0,0813	0,0795	0,0844	0,0803

Průměrně tedy potřebovala sériová implementace k převedení obrázku na odstíny šedé $0,0832$ s.

3.1.2 Výsledky paralelní implementace

Výsledky časové náročnosti paralelní implementace kódu (A.2) jsou uvedeny v následující tabulce 3.2.

Tab. 3.2: Doba trvání paralelní implementace

Pořadí měření	[-]	1	2	3	4	5
Naměřený čas	[s]	0,0908	0,0911	0,0892	0,0923	0,0936
Pořadí měření	[-]	6	7	8	9	10
Naměřený čas	[s]	0,0940	0,0901	0,0915	0,0913	0,0904

Průměrný čas, který pro převedení obrázku na odstíny šedé potřebovala paralelní implementace, je $0,0914$ s.

3.2 Implementace paralelizace na dlouhém kódu

Pro otestování časové úspory při použití paralelizace dlouhého kódu mi byly poskytnuty MATLABové zdrojové kódy implementace extrapolačního algoritmus K–SVD 2.2.1. Z nich byly podle kritérií (viz 1.4.2) pro využití paralelizace vybrány funkce `OMP_Impaint`, `Extrapolate_OMP`, `Extrapolate_Dictionary`.

3.2.1 Výsledky sériové implementace

Naměřená časová náročnost sériového kódu.

Tab. 3.3: Doba běhu sériového kódu

Pořadí měření	[-]	1	2	3	4	5
Naměřený čas	[s]	50,469	47,404	49,995	55,009	54,675
Pořadí měření	[-]	6	7	8	9	10
Naměřený čas	[s]	58,381	49,864	50,024	49,360	49,948

Sériová implementace K–SVD algoritmu potřebovala k běhu průměrně $51,513$ s.

3.2.2 Výsledky paralelní implementace

Výsledky časové náročnosti s použitím paralelizovaných funkcí `OMP_Impaint`, `Extrapolate_OMP`, `Extrapolate_Dictionary`.

Tab. 3.4: Doba běhu paralelizovaného kódu

Pořadí měření	[-]	1	2	3	4	5
Naměřený čas	[s]	16,077	16,027	16,039	15,942	15,935
Pořadí měření	[-]	6	7	8	9	10
Naměřený čas	[s]	16,042	15,994	16,024	15,976	15,955

Při použití paralelizovaných funkcí byla průměrná délka zpracování $16,001$ s. Došlo tedy k průměrnému zrychlení o $35,511$ s, což odpovídá časové úspoře $68,937$ %.

4 ZÁVĚR

Ve svojí práci jsem prozkoumal možnosti paralelizace výpočtů v MATLABu a uplatnil je nejprve na jednoduchý převod barevného snímku na černobílý dle rovnice (2.1). Předpokládal jsem zrychlení výpočtů po převedení sekvenčního kódu na paralelní kód. Z tabulek měření časové náročnosti jednotlivých implementací 3.1 a 3.2 je vidět, že tento předpoklad se nenaplnil. Příčinou je pravděpodobně implementace na příliš krátkém kódu, kde vyšší počáteční režie, kterou paralelní programy vyžadují při svojí inicializaci, převýšila její výhody.

Paralelizace proto byla podruhé aplikována na dlouhý kód algoritmu K–SVD a na funkce splňující kritéria pro převod do paralelního kódu. Jak vidíme v tabulkách časové náročnosti v sekci 3.2, potvrdilo se, že skryté počáteční nároky paralelizace jsou na krátkém kódu příliš vysoké a mohou vést ke zpomalení. Při použití paralelních funkcí jsme dosáhli zrychlení o $68,937$ %.

LITERATURA

[1] AHARON, M.; ELAD, M.; BRUCKSTEIN, A. *K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation* IEEE Transactions on Signal Processing, roč. 54, č. 11, s. 4311–4322, 2006

[2] BRÄUNL, T. *Parallel image processing*. Berlin: Springer, 2010, ix, 203 s. ISBN 978-3-642-08679-3.

[3] HORÁK, J. *Von Neumannova a Harvardská architektura*. [online]. [cit. 8. 3. 2013]. Dostupné z URL: <http://lsd.spsejecna.net/web/beranek/I3B/HorakJan_Von%20Neumannova%20architektura.pdf>.

[4] HRBÁČEK, R.; RAJMIC, P.; VESELÝ, V.; ŠPIŘÍK, J. *Řídké reprezentace signálů: Úvod do problematiky.* Elektrorevue - Internetový časopis (http://www.elektrorevue.cz), 2011, roč. 2011, č. 50, s. 1-10. ISSN: 1213- 1539.

[5] *Intel Reinvents Transistors Using New 3-D Structure*. Intel UK Newsroom. 2011. [online]. [cit. 6. 3. 2013]. Dostupné z URL: <http://newsroom.intel.com/community/en_uk/blog/2011/05/04/intel-reinvents-transistors-using-new-3-d-structure>.

[6] *ITRS activities 2010*. International technology roadmap for semiconductors. 2010. [online]. [cit. 7. 3. 2013]. Dostupné z URL: <http://www.itrs.net/Links/2010ITRS/2010Update/ToPost/2010Tables_ORTC_ITRS.xls>.

[7] KEPNER, J. *Parallel MATLAB for multicore and multinode computers*. Philadelphia: Society for Industrial and Applied Mathematics, 2009, xxv, 253 s. ISBN 978-0-898716-73-3.

[8] KEPNER, Jeremy. BLISS, Nadya. *pMatlab: Parallel Matlab Toolbox v2.0.1*. [online]. [cit. 10. 3. 2013]. Dostupné z URL: <http://www.ll.mit.edu/mission/isr/pmatlab/pmatlab.html>.

[9] *MATLAB Functions in Parallel Computing Toolbox*. The MathWorks, Inc. [online]. [cit. 22. 3. 2013]. Dostupné z URL: <http://www.mathworks.com/help/distcomp/functionlist.html>.

[10] *Microsoft Developer Network*. Microsoft. [online]. [cit. 15. 3. 2013]. Dostupné z URL: <http://msdn.microsoft.com/en-us/library/ff963553.aspx>.

[11] MOORE, G. E. *Cramming more components onto integrated circuits*. Electronics, 1965, č. 8. [online]. [cit. 6. 3. 2013]. Dostupné z URL:

<http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf>.

[12] *Parallel Computing Toolbox*. The MathWorks, Inc. [online]. [cit. 21. 5. 2013]. Dostupné z URL: <http://www.mathworks.com/products/parallel-computing/index.html>.

[13] PETERKA, Jiří. *Von Neumannova architektura.*. [online]. [cit. 9. 3. 2012]. Dostupné z URL: <http://www.earchiv.cz/a93/a321c120.php3>.

[14] *Star-P® Software Development Kit (SDK) Tutorial and Reference Guide*. Interactive Supercomputing Inc., Massachusetts institute of technology. [online]. [cit. 20. 3. 2013]. Dostupné z URL: <http://static.msi.umn.edu/tutorial/scientificcomp/Starp_UserGuide.pdf>.

[15] STOKES, Jon. *SIMD architectures.*. [online]. [cit. 9. 4. 2013]. Dostupné z URL: <http://arstechnica.com/features/2000/03/simd/>.

[16] *Top500 List - November 2012*. TOP500.org. [online]. [cit. 20. 5. 2013]. Dostupné z URL: <http://top500.org/lists/2012/11/>.

[17] ŠPIŘÍK, J.; ZÁTYIK, J.; LOCSI, L. *Image extrapolation using K-SVD algorithm. In 36th International Conference on Telecommunications and Signal Processing (TSP 2013).* 2013. ISBN 978-1-4799-0404-4.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

ALU aritmeticko–logická jenodtka

GPU Graphics processing unit

K–SVD K–times Singular Value Decomposition

MIMD multiple instruction multiple data

MIT Massachusetts Institute of Technology

N_P Počet běžících instancí nebo kopií programu

OMP Orthogonal Matching pursuit

PE processing element

P_{ID} Unikátní identifikační číslo jednotlivé instance programu

SIMD single instruction multiple data

SISD single instruction single data

UCSB University of California, Santa Barbara

SEZNAM PŘÍLOH

A Ukázky kódu **33**

A.1 Sériová implementace převodu obrázku do odstínů šedé 33

A.2 Paralelní implementace převodu obrázku do odstínů šedé 34

A.3 Extrapolate OMP 35

A.4 OMP Inpaint 39

A.5 Extrapolate Dictionary 43

A.6 Převod z pole buněk do jednorozměrného pole 53

A.7 Vytvoření pole koeficientů pro další použití 54

A UKÁZKY KÓDU

A.1 Sériová implementace převodu obrázku do odstínů šedé

```
1  function [resultpicture] = easygray(picture)
2  % jednoduchy prevod rgb obrazku na odstiny sede
3  %   Funkce prevadi obrazek z barevneho do odstinu sede pomoci ...
       vzorce Y'= 0.2989*R + 0.5870*G + 0.1141*B;
4      temp=uint8(0);
5      temp(512,512)=temp;
6      tic;
7      for(j=1:512)
8         for(k=1:512)
9            temp(j,k)=0.2989*picture(j,k,1)+0.5870*picture(j,k,2)+
10           0.1141*picture(j,k,3);
11        end
12     end
13
14     resultpicture=temp;
15     toc;
16 end
```

A.2 Paralelní implementace převodu obrázku do odstínů šedé

```
1  function [resultpicture] = easygrayparallel(picture)
2  % jednoduchy prevod rgb obrazku na odstiny sede
3  %   Funkce prevadi obrazek z barevneho do odstinu sede pomoci ...
       NTSC. Prevod
4  %   je dan vzorcem Y'= 0.2989*R + 0.5870*G + 0.1141*B;
5
6      resultpicture=uint8(0);
7      resultpicture(512,512)=resultpicture;
8
9      clear temppic;
10     %matlabpool local 6;
11     tic;
12     parfor j=1:512
13         temppic=zeros(1,512);
14         for k=1:512
15             temppic(k)=0.2989*picture(j,k,1)+0.5870*picture(j,k,2)+
16             0.1141*picture(j,k,3);
17         end
18         resultpicture(j,:)=temppic;
19     end
20     toc;
21     %matlabpool close;
22 end
```

A.3 Extrapolate OMP

```
1  function [resX , resA] = Extrapolate_OMP(D , X , param)
2  % Run Matching Pursuit
3  %
4  % Inputs :
5  % D     : dictionary (normalized columns)
6  % X     : set of vectors to run on (each column is one signal)
7  % param : stopping condition, containing at least one of these rows
8  %         * 'errorGoal' and 'noiseSig'
9  %         * 'maxAtoms'
10 %
11 % Outputs :
12 % resX  : The result vectors
13 % resA  : Sparse coefficient matrix
14 
15 % Get parameters and allocate result
16 dim = size(D , 1);
17 nAtoms = size(D , 2);
18 nSignals = size(X , 2);
19 
20 % determine stopping criteria
21 testErrorGoal = false; errorGoal = inf;
22 if isfield(param , 'errorGoal'),
23     testErrorGoal = true;
24 
25     % Compute the actual error goal, and account for noise and ...
           signal length
26     errorGoal = param.errorGoal * dim * (param.noiseSig.^2);
27 end;
28 testMaxAtoms = false; maxAtoms = nAtoms;
29 if isfield(param , 'maxAtoms'), testMaxAtoms = true; maxAtoms = ...
       param.maxAtoms; end;
30 if (¬testErrorGoal) && (¬testMaxAtoms), error('At least one ...
       stopping criterion is needed!'); end;
31 
32 % Allocate vectors to insert coefficients into
33 % We keep them as triplets (signalInd, atomInd, coeffVal)
34 % In the end, we will construct a sparse matrix from them, as ...
       this is more efficient
35 allSignalsListPAR = zeros(1,nSignals);
36 allIndsListPAR = cell(1,nSignals);
37 allCoeffsListPAR = cell(1,nSignals);
38 
39 % Compute DtD and DtX
```

```
40 DtD = D' * D;
41 DtX = D' * X; % This might not work for a large number of signals.
42              % It is usedful to break X into groups of signals ...
                    in that case
43              % Alternatively, this can be computed for each ...
                    signal, however, this is slower
44 sigSumSqrs = sum(X.^2 , 1);
45
46 % Run loop on signals
47 parfor cSignal = 1 : nSignals
48
49     % get current signal - get its inner products with the ...
           dictionary directly (D^T x)
50     initInnerProductsList = DtX(: , cSignal);
51     currInnerProductsList = initInnerProductsList;
52
53     % init the residual size counter
54     residSumSqrs = sigSumSqrs(cSignal);
55
56     % This is used for updating the resiudal norm at each stage
57     prev_Δ = 0;
58
59     % Make sure the initial signal in itself has enough energy,
60     % otherwise the zero signal is returned
61     if ((testErrorGoal) && (residSumSqrs < errorGoal))
62         continue; % simply move on to the next signal
63     end
64
65     % Initialize indices vectors
66     indsList = [];
67     card = 0;
68
69     % Repeat as long as stopping condition isn't filled
70     while 1
71
72         % Update cardinality
73         card = card + 1;
74
75         % Find the index of the largest absolute inner product
76         maxProjInd = ...
77 find(max(abs(currInnerProductsList))==abs(currInnerProductsList),1);
78
79         % If this is the first atom, keep its projection
80         if (card == 1)
81             coeffsList = currInnerProductsList(maxProjInd);
82             indsList = maxProjInd;
```

```
83          else
84              % If not the first atom, do least-squares (LS) over ...
                    all atoms in the representation
85              indsList = [indsList maxProjInd];
86              LS_LHS = DtD(indsList , indsList); % The left hand ...
                    side of the LS equation to solve
87              LS_RHS = initInnerProductsList(indsList);
88              coeffsList = LS_LHS \ LS_RHS;
89          end
90
91          % Update the inner products list
92          beta = DtD(: , indsList) * coeffsList;
93          currInnerProductsList = initInnerProductsList - beta;
94
95          % Check if we can stop running
96          if testErrorGoal % Error Treshold
97
98              % We only need to update the residual computation if ...
                    the stopping criterion is the error
99              curr_Δ = sum(coeffsList .* beta(indsList));
100             residSumSqrs = residSumSqrs + prev_Δ - curr_Δ;
101             prev_Δ = curr_Δ;
102
103             if residSumSqrs < errorGoal, break; end;
104         end
105
106         if testMaxAtoms % Cardinality Threshold
107             if (card ≥ maxAtoms), break; end;
108         end
109
110     end
111
112     % Assign results
113     allCoeffsListPAR{cSignal}=coeffsList;
114     allIndsListPAR{cSignal}=indsList;
115     allSignalsListPAR(cSignal)=cSignal;
116 end
117 % close(h);
118
119 totNcoeffs=(numel(allSignalsListPAR)-sum(allSignalsListPAR==0))*20;
120 allCoeffsList=extractList(allCoeffsListPAR,allSignalsListPAR);
121 allIndsList=extractList(allIndsListPAR,allSignalsListPAR);
122 allSignalsList=expandSignalsList(allSignalsListPAR);
123
124 % if (nSignals > 1000), fprintf('\n'); end;
125
```

```
126 % Construct the sparse matrix
127 resA = sparse(...
128     allIndsList(1:totNcoeffs) , allSignalsList(1:totNcoeffs) , ...
        allCoeffsList(1:totNcoeffs) , ...
129     nAtoms , nSignals);
130
131 % Create the output signals
132 resX = D * resA;
133
134 % Finished
135 return;
```

A.4 OMP Inpaint

```
1  function [resX , resA] = OMP_Inpaint(D , X , Masks, param)
2  % Run Orthogonal Matching Pursuit
3  %
4  % Inputs :
5  % D      : dictionary (normalized columns)
6  % X      : set of vectors to run on (each column is one signal)
7  % param : * stopping condition : ('errorGoal' and 'noiseSig') or ...
       'maxAtoms'
8  %          * 'IgnoreUnmasked' (Optional) : 1 to ignore unmasked ...
       patches, 0 to do them (default: 0)
9  %
10 %
11 % Outputs :
12 % resX  : The result vectors
13 % resA  : Sparse coefficient matrix
14
15 % Get parameters and allocate result
16 dim = size(D , 1);
17 nAtoms = size(D , 2);
18 nSignals = size(X , 2);
19 if ¬isfield(param , 'IgnoreUnmasked'), param.IgnoreUnmasked = 0; ...
       end;
20
21 % determine stopping criteria
22 testErrorGoal = false; errorGoal = inf;
23 if isfield(param , 'errorGoal'),
24     testErrorGoal = true;
25
26     % Compute the actual error goal, and account for noise and ...
           signal length
27     errorGoal = param.errorGoal * dim * (param.noiseSig.^2);
28 end;
29 testMaxAtoms = false; maxAtoms = nAtoms;
30 if isfield(param , 'maxAtoms'), testMaxAtoms = true; maxAtoms = ...
       param.maxAtoms; end;
31 if (¬testErrorGoal) && (¬testMaxAtoms), error('At least one ...
       stopping criterion is needed!'); end;
32
33 % Allocate vectors to insert coefficients into
34 % We keep them as triplets (signalInd, atomInd, coeffVal)
35 % In the end, we will construct a sparse matrix from them, as ...
       this is more efficient
36 allSignalsListPAR = zeros(1,nSignals);
```

```
37 allIndsListPAR = cell(1,nSignals);
38 allCoeffsListPAR = cell(1,nSignals);
39
40 % See which patches we need to ignore, if we do
41 if param.IgnoreUnmasked
42     sumUnmasked = sum(Masks , 1);
43     patchesToIgnore = sumUnmasked == dim;
44 else
45     patchesToIgnore = [];
46 end
47
48 % Run loop on signals
49 parfor cSignal = 1 : nSignals
50
51     % process one signal
52
53     % Get current mask
54     m = Masks(: , cSignal);
55     vm = find(m);
56
57     % Ignore this patch if it is not masked at all
58     if (param.IgnoreUnmasked && (sum(m) == dim))
59         continue;
60     end
61
62     % Get current signal
63     x = X(vm , cSignal);
64
65     % Take the needed dictionary rows
66     currD = D(vm , :);
67     atomsNormFacts = sqrt(sum(currD.^2 , 1));
68     currD = currD ./ repmat(atomsNormFacts , [size(currD , 1) 1]);
69
70     % Update error goal
71     currErrorGoal = errorGoal * length(vm) / dim;
72
73     % Initialize coefficient and indices vectors
74     residual = x;
75     coeffsList = zeros(dim , 1);
76     indsList = zeros(1 , dim);
77     card = 0;
78
79     % Make sure the initial signal in itself has enough energy,
80     % otherwise the zero signal is returned
81     if ((testErrorGoal) && (sum(residual.^2) < currErrorGoal))
82         continue; % simply move on to the next signal
```

```
83      end
84
85      % Repeat as long as stopping condition isn't filled
86      while 1
87
88          % Update cardinality
89          card = card + 1;
90
91          % Compute projections
92          proj = currD' * residual;
93
94          % Find the index of the largest absolute inner product
95          maxProjInd = find(max(abs(proj))==abs(proj),1);
96
97          % Add this new atom to the list
98          indsList(card) = maxProjInd;
99
100         % If this is the first atom, keep its projection
101         if (card == 1)
102             coeffsList(card) = proj(maxProjInd);
103         else
104             % If not the first atom, do least-squares (LS) over ...
                    all atoms in the representation
105             coeffsList(1 : card) = currD(: , indsList(1 : card)) ...
                    \ x;
106         end
107
108         % Compute new residual
109         residual = x - currD(: , indsList(1 : card)) * ...
                coeffsList(1 : card);
110
111         % Check if we can stop running
112         if testErrorGoal % Error Treshold
113             residNorm = sum(residual.^2);
114             if residNorm < currErrorGoal, break; end;
115         end
116
117         if testMaxAtoms % Cardinality Threshold
118             if (card ≥ maxAtoms), break; end;
119         end
120
121     end
122
123     % We need to re-modify the coefficients,
124     % to undo the effects of the dictionary normalization
125     if (card > 0)
```

```
126         coeffsList(1 : card) = coeffsList(1 : card) ./ ...
                atomsNormFacts(indsList(1 : card))';
127     end
128
129     % Assign results
130     allCoeffsListPAR{cSignal}=coeffsList;
131     allIndsListPAR{cSignal}=indsList;
132     allSignalsListPAR(cSignal)=cSignal;
133 end
134 %close(h);
135
136 totNcoeffs=(numel(allSignalsListPAR)-sum(allSignalsListPAR==0))*20;
137 allCoeffsList=extractList(allCoeffsListPAR,allSignalsListPAR);
138 allIndsList=extractList(allIndsListPAR,allSignalsListPAR);
139 allSignalsList=expandSignalsList(allSignalsListPAR);
140
141 % Construct the sparse matrix
142 resA = sparse(...
143     allIndsList(1:totNcoeffs) , allSignalsList(1:totNcoeffs) , ...
            allCoeffsList(1:totNcoeffs) , ...
144     nAtoms , nSignals);
145
146 % Create the output signals
147 resX = D * resA;
148 resX(: , patchesToIgnore) = X(: , patchesToIgnore);
149
150 % Finished
151 return;
```

A.5 Extrapolate Dictionary

```
1  function [resDict , allDicts] = ...
       Extrapolate_Dictionary(trainPatches, param)
2  % Train a dictionary using either K−SVD or MOD
3  %
4  % Inputs :
5  % trainPatches : The set of patches to train on
6  % param        : various parameters, continaing the fields ...
       (fields with defaul indicated are optional)
7  %                 * 'method'          : 'KSVD' or 'MOD'
8  %                 * (at least) One of ('errorGoal' & 'noiseSig') ...
       or 'maxAtoms' :
9  %                                       stopping condition for ...
       the OMP
10 %                 * 'nAtoms'          : number of atoms in the ...
       dictionary
11 %                 * 'nIterations'     : number of dictionary ...
       update iterations
12 %                 * 'initType'        : 'RandomPatches' to take a ...
       random set of patches,
13 %                                       'DCT' for an overcomplete ...
       DCT dictionary (assumes patch is square)
14 %                                       'Input' an initial ...
       dictionary is given
15 %                 * 'initDict'        : Only if param.initType == ...
       'Input' − the initial dictionary to use
16 %                 * 'maxIPforAtoms'   : The maximal allowed inner ...
       profuct between two atoms
17 %                                       (if two atoms are created ...
       with larger IP, one is replaced).
18 %                                       Default: 0.99
19 %                 * 'minTimesUsed'    : The minimal number of ...
       times an atom should have a meaningful coefficient.
20 %                                       If less or equal to that, ...
       it is replaced
21 %                                       Default: 3
22 %                 * 'meaningfulCoeff' : The smallest value that a ...
       coefficient is considered to be different than 0.
23 %                                       Default: 10^−7
24 %                 * 'useLessAtoms'    : NOT YET IMPLEMENTED
25 %                                       Use for several ...
       iterations less than all the atoms.
26 %                                       This is a vector, with ...
       numbers for each iteration
```

```
27 %                                     If it is shorter, for ...
    later iterations all patches are used
28 %                                     A number 0.1 means about ...
    0.1 of the patches are used
29 %               * 'showDictionary'   : After how many iterations ...
    show the dictionary (0 to no-show)
30 %                                     Default: 0
31 %               * 'patchSize'        : Needed for showing the ...
    dictionary
32 %               * 'trueDictionary'   : Optional. Needed to ...
    estimate how close the recovered dictionary is to true one
33 %               * 'atomRecoveredThresh' : Needed if ...
    'trueDictionary' is provided.
34 %                                     Atom from trueDictionary ...
    is considered to be recovered
35 %                                     if there's an atom in the ...
    current dictionary,
36 %                                     the absolute inner ...
    product of these atoms is above this
37 %               * 'truePatches'      : Optional. The ground ...
    truth patches (for computing representation error at each
38 %                                     iteration)
39 %               * 'imageData'        : Optional struct, if the ...
    dictionary is used for image denoising.
40 %                                     If avaliable, uses the ...
    dictionary at each stage (except the last) to
41 %                                     create an image.
42 %                                     If the ground truth image ...
    is also avaliable, computes the PSNR.
43 %                                     It should have the fields:
44 %                                     'patchSize' (the size of ...
    the patch [h w])
45 %                                     'imageSize' (the size of ...
    the original image [h w])
46 %                                     'withOverlap' (1 or 0, ...
    wether patches overlap. If doens't exist, assumes overlap)
47 %                                     'showImage' wether to ...
    show the image in each iteration
48 %                                     'groundTruthIm' the real ...
    image
49 %
50 %
51 % Outputs :
52 % ResDict        : The final dictionary
53 % AllDicts       : A cell array with the dictionaries for each ...
    iteration
```

```
54 %                   (first entry is initial, second entry is after ...
       first iteration, etc.)
55
56 % Reset random number generator
57 randn('state',0);
58
59 % Get parameters
60 dim = size(trainPatches , 1);
61 nAtoms = param.nAtoms;
62 nPatches = size(trainPatches , 2);
63
64 % Default values for some parameters
65 if ¬isfield(param , 'maxIPforAtoms'), param.maxIPforAtoms = ...
       0.99; end;
66 if ¬isfield(param , 'minTimesUsed'), param.minTimesUsed = 3; end;
67 if ¬isfield(param , 'meaningfulCoeff'), param.meaningfulCoeff = ...
       10^-7; end;
68 if ¬isfield(param , 'showDictionary'), param.showDictionary = 1; ...
       end;
69 if param.showDictionary && ¬isfield(param , 'patchSize'),
70     error('Must specify patch size for showing dictionary');
71 end;
72
73
74 % Initialize dictionary
75 switch param.initType
76     case 'RandomPatches'
77         % Select a random sub-set of the patches
78         p = randperm(nPatches);
79         currDict = trainPatches(: , p(1 : nAtoms));
80
81     case 'DCT'
82         % Compute the over-complete DCT dictionary, and remove ...
               unnecessary columns and rows
83         DCTdict = Build_DCT_Overcomplete_Dictionary(nAtoms , ...
               ceil(sqrt(dim)) .* [1 1]);
84         currDict = DCTdict(1 : dim , 1 : nAtoms);
85
86     case 'Input'
87         % Use a given dictionary
88         currDict = param.initDict;
89
90     otherwise
91         error('Unknown dictionary initialization method : %s' , ...
               param.initType);
92 end
```

```
93
94  % Normalize columns of the dictionary
95  currDict = currDict ./ repmat(sqrt(sum(currDict.^2, 1)) , ...
        size(currDict , 1) , 1);
96
97  % store the initial dictionary
98  allDicts = cell(1 , param.nIterations+1);
99  allDicts{1} = currDict;
100
101 % If ground truth is provided, compute the noisy's quality
102 if isfield(param , 'truePatches')
103     signalErrors = param.truePatches - trainPatches;
104     meanSignalsError = mean(sum(signalErrors.^2 , 1));
105     fprintf('mean error of noisy signals %02.2f\n' , ...
            meanSignalsError);
106 end
107
108 % Run loops as required
109 parfor itr = 1 : param.nIterations
110
111     currDictPAR = currDict;
112
113     if isfield(param , 'useLessAtoms') && ...
            length(param.useLessAtoms) ≥ itr
114         takePatchesInds = unique(round(linspace(1 , ...
                size(trainPatches,2) , param.useLessAtoms(itr) * ...
                size(trainPatches,2))));
115     else
116         takePatchesInds = [1 : size(trainPatches,2)];
117     end
118     currTrainPatches = trainPatches(: , takePatchesInds);
119     usingAllPatches = size(currTrainPatches,2) == ...
            size(trainPatches , 2);
120
121     % Do sparse coding - we need the reconstructed patches for ...
            the residual
122     [currConstructedPatches , currCoeffs] = ...
            Extrapolate_OMP(currDictPAR , currTrainPatches , param);
123
124     % Compute average cardinality or representation error, and ...
            print information
125 %   if isfield(param , 'errorGoal')
126 %       meanCard = full(sum(abs(currCoeffs(:)) > ...
        param.meaningfulCoeff) / length(takePatchesInds));
127 %       disp(['       Iteration ',num2str(itr),' ==> Average ...
        Cardinality ',num2str(meanCard)]);
```

```
128 %   end
129 %
130 %   if isfield(param , 'maxAtoms')
131 %       resids = currTrainPatches - currConstructedPatches;
132 %       meanError = mean(sum(resids.^2 , 1));
133 %       fprintf('      mean representation error (compared to ...
        training signals) %02.2f\n' , meanError);
134 %   end
135 %
136 %   % If the true signals are provided, compute the L2 error of ...
        their representation
137 %   if isfield(param , 'truePatches')
138 %       signalErrors = param.truePatches(: , takePatchesInds) - ...
        currConstructedPatches;
139 %       meanSignalsError = mean(sum(signalErrors.^2 , 1));
140 %       fprintf('      mean recovery error (compared to true ...
        signals) %02.2f\n' , meanSignalsError);
141 %   end
142
143
144     % If image data is provided, reconstruct image and compute PSNR
145     if isfield(param , 'imageData') && usingAllPatches
146
147         % Create image
148         currIm = ...
                Average_Overlapping_Patches(currConstructedPatches, ...
149             size(param.imageData.imageSize) , ...
                    param.imageData.patchSize);
150
151         % If ground truth is provided, compute the estimate's ...
                quality
152         if isfield(param.imageData , 'groundTruthIm')
153             [currPSNR , currL2] = ...
                    Compute_Error_Stats(param.imageData.groundTruthIm ...
                    , currIm);
154             fprintf('Current IMAGE PSNR %02.4f, L2 %03.4f\n' , ...
                    currPSNR , currL2);
155         end
156
157         % Display image if required
158         if param.imageData.showImage
159             figure; imshow(currIm);
160             if isfield(param.imageData , 'groundTruthIm')
161                 title(sprintf('Iteration %d , PSNR = %02.4' , ...
                        itr - 1, currPSNR));
162             else
```

```
163                 title(sprintf('Iteration %d ' , itr - 1));
164             end
165         end
166 
167 
168 
169     end
170 
171     % Only here the new training iteration starts -
172     % up until here we used the previous dictionary
173     % fprintf('Iteration %d              \n' , itr);
174 
175 
176     % Update dictionary using requested method
177     switch param.method
178         case 'MOD'
179 
180             % Compute the new dictionary - at once
181             % The addition of (eye(nAtoms) * 10^-7) is for ...
                    regularization
182             % newDict = (trainPatches * currCoeffs') * ...
                    inv(currCoeffs * currCoeffs' + (eye(nAtoms) * ...
                    10^-7));
183 
184             % Matlab claims the previous equation is ...
                    inefficient. This should be equivalent but faster
185             newDict = (currTrainPatches * currCoeffs') / ...
                    (currCoeffs * currCoeffs' + (eye(nAtoms) * 10^-7));
186 
187 
188             % Remove zero vectors by random vectors
189             zeroAtomsInds = find(sum(abs(newDict) , 1) < 10^-10);
190             newDict(: , zeroAtomsInds) = randn(dim , ...
                    length(zeroAtomsInds));
191 
192             % Normalize atoms to norm 1
193             newDict = newDict ./ repmat(sqrt(sum(newDict.^2, 1)) ...
                    , size(newDict , 1) , 1);
194 
195         case 'KSVD'
196 
197             % Compute one new atom at a time
198             % This is incremental (i.e., use previou updated ...
                    atoms from current iteration too)
199             newDict = currDictPAR;
200 
```

```
201             % Run a loop on atoms
202             % fprintf('      Updating Atoms\n ' );
203             for atomInd = 1 : nAtoms
204 
205                 % if mod(atomInd , 10) == 0, fprintf('%d ' , ...
                        atomInd); end;
206 
207                 % Find all patches using it
208                 currPatchsInds = find(currCoeffs(atomInd , :));
209 
210                 % Get all coefficients for patches that use the atom
211                 currPatchsCoeffs = currCoeffs(: , currPatchsInds);
212 
213                 % Set to zero the coefficient of the current ...
                        atom for each patch
214                 currPatchsCoeffs(atomInd , :) = 0;
215 
216                 % Compute the residuals for all signals
217                 resids = currTrainPatches(: , currPatchsInds) − ...
                        newDict * currPatchsCoeffs;
218 
219                 % Use svd to determine the new atom, and the new ...
                        coefficients
220                 [newAtom , singularVal, betaVec] = svds(resids , 1);
221 
222                 newDict(: , atomInd) = newAtom; % Insert new atom
223                 currCoeffs(atomInd , currPatchsInds) = ...
                        singularVal * betaVec'; % Use coefficients ...
                        for this atom
224             end
225             % fprintf('\n');
226 
227         otherwise
228             error('Unknown dictionary update method %S' , method);
229     end
230 
231     % compute the residual of each signal
232     resids = currTrainPatches − newDict * currCoeffs;
233 
234     % To improve the dictionary, we now do 2 things:
235     % 1. Remove atoms that are rarely used, and replace them by ...
            a random entry
236     % 2. Remove atoms that have very similar atoms in the ...
            dictionary (i.e., large inner product)
237 
238     % Compute the representation error of each signal
```

```
239     sigRepErr = sum(resids .^ 2 , 1);
240     dictIP = newDict' * newDict;
241     dictIP = dictIP .* (1 - eye(size(dictIP))); % Zero the ...
            diagonal - the IP of the atom with itself, so it won't ...
            bother us
242
243     % Run on each atom, and make the checks
244     nReplacesAtoms = 0;
245     for cAtom = 1 : nAtoms
246
247         maxIPwithOtherAtom = max(dictIP(cAtom , :));
248         numTimesUsed = sum(abs(currCoeffs(cAtom , :)) > ...
                param.meaningfulCoeff);
249
250         if ((maxIPwithOtherAtom > param.maxIPforAtoms) || ...
                (numTimesUsed ≤ param.minTimesUsed))
251
252             nReplacesAtoms = nReplacesAtoms + 1;
253
254             % Replace the atom with the signal that is worst ...
                    represented
255             worstSigInd = find(max(sigRepErr)==sigRepErr,1);
256             worstSigInd = worstSigInd(1);
257             newAtom = currTrainPatches(: , worstSigInd);
258             newAtom = newAtom / norm(newAtom);
259             newDict(: , cAtom) = newAtom;
260
261             % Update the inner products matrix and the ...
                    representation errors matrix
262             sigRepErr(worstSigInd) = 0; % Since it now has an ...
                    atom for it.
263             newIPsForAtom = newDict' * newAtom;
264             newIPsForAtom(cAtom) = 0; % The inner product with ...
                    itself, which we want to ignore
265             dictIP(cAtom , :) = newIPsForAtom';
266             dictIP(: , cAtom) = newIPsForAtom;
267         end
268     end
269     % fprintf('      %d atoms replaced for stability\n' , ...
            nReplacesAtoms);
270
271
272
273     % If we are provided with a ground-truth dictionary, we do ...
            two things:
```

```
274     % Try to order the atoms in the estimated dictionary in the ...
            same order as the ground-truth one
275     % See how many of the true atoms have been found
276     if isfield(param , 'trueDictionary')
277
278         % We try to re-order the atoms in the following manner:
279         % First, we compute the inner-products between ever true ...
                atom and every estimate atom
280         % We then go and find the largest IP in this matrix, and ...
                determine
281         % what true atom and estimated atom achieved it.
282         % We then re-order the estimated atom to be in the same ...
                place as the original atom
283         % and zero both these column and row so both atoms ...
                aren't selected again
284
285         % Compute the absolute IPs
286         true_CurrDict_AbsIPs = abs(param.trueDictionary' * newDict);
287
288         % Allocate space for the order transformation
289         atomTrns = zeros(1 , nAtoms);
290
291         % Run for as many iterations as atoms
292         for cFound = 1 : nAtoms
293
294             % Find the maximal IP
295             [bestTrueAtomInd , bestEstAtomInd] = ...
                    find(true_CurrDict_AbsIPs == ...
                    max(true_CurrDict_AbsIPs(:)));
296             bestEstAtomInd = bestEstAtomInd(1); bestTrueAtomInd ...
                    = bestTrueAtomInd(1); % In case we found more ...
                    than one
297
298             % Log this transformation
299             atomTrns(bestTrueAtomInd) = bestEstAtomInd;
300
301             % Make sure the same atoms aren't selected again
302             true_CurrDict_AbsIPs(bestTrueAtomInd , :) = 0;
303             true_CurrDict_AbsIPs(: , bestEstAtomInd) = 0;
304
305         end
306
307         % Perform the re-ordering
308         newDict = newDict(: , atomTrns);
309
310         % In some cases, atoms have reverse polarities.
```

```
311            % So, we want to change the polarities of the estimated ...
                   atoms to match
312            % that of the true atoms
313            afterTrnsIPs = sum(newDict .* param.trueDictionary);
314            signFactor = sign(afterTrnsIPs); signFactor(signFactor ...
                   == 0) = 1;
315            newDict = newDict .* repmat(signFactor , size(newDict,1) ...
                   , 1);
316
317            % Using the IPs after the transformation to
318            % detect if the true atoms are recovered
319            isAtomDetected = abs(afterTrnsIPs) ≥ ...
                   param.atomRecoveredThresh;
320            fprintf('      Percentage of found atoms = %03.2f\n' , ...
                   100 * mean(isAtomDetected));
321
322        end
323
324
325        % Update the dictionary list
326        allDicts{itr + 1} = newDict;
327
328        % Show the dictionary if required
329    %   if (param.showDictionary > 0) && (mod(itr , ...
           param.showDictionary) == 0)
330    %       if itr==1, figure; end; % Not open new figure every time
331    %       Show_Dictionary(currDict);
332    %       title(sprintf('Dictionary after iteration %d' , itr));
333    %       drawnow;
334    %   end
335    %
336    %   drawnow;
337    end
338
339    % Return the last dictionary
340    resDict = allDicts{param.nIterations+1};
341
342    % Finished
343    return;
```

A.6 Převod z pole buněk do jednorozměrného pole

```
1  function [ allCoeffsListOUT ] = extractList( ...
       allCoeffsListIN,allsignalList )
2  %Funkce vybere podle nenulovych prvku v allsignalList ...
       odpovidajici bunky z allCoeffsListIN
3  %
4
5      listSize = numel(allsignalList);
6      allCoeffsListOUT = zeros(1,listSize*20);
7      myCard = 20;
8      indexPointer = 0;
9
10     for i=1:listSize
11         if allsignalList(i)≠0
12             allCoeffsListOUT(indexPointer+(1:myCard)) = ...
                   allCoeffsListIN{i}(1:myCard);
13             indexPointer = indexPointer+myCard;
14         end
15     end
16
17 end
```

A.7 Vytvoření pole koeficientů pro další použití

```
1  function [ allSignalListOUT ] = expandSignalsList( ...
       allSignalListIN )
2  %Rozsiri pole allSignalListIN na rozmer, ktery pro vstup ...
       vyzaduji dalsi funkce
3  %
4      allSignalListOUT = zeros(1,numel(allSignalListIN)*20);
5      myCard = 20;
6      indexPointer = 0;
7
8      for i=1:numel(allSignalListIN)
9          if allSignalListIN(i)≠0
10             allSignalListOUT(indexPointer+(1:myCard)) = ...
                   allSignalListIN(i);
11             indexPointer = indexPointer + myCard;
12         end
13     end
14 end
```