**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA


# PARALELNÍ A DISTRIBUOVANÉ ZPRACOVÁNÍ ROZSÁHLÝCH TEXTOVÝCH DAT

**PARALLEL AND DISTRIBUTED PROCESSING OF LARGE TEXTUAL DATA**

**DIPLOMOVÁ PRÁCE**
MASTER'S THESIS


**AUTOR PRÁCE** — **Bc. MARTIN MATOUŠEK**
AUTHOR

**VEDOUCÍ PRÁCE** — **Doc. RNDr. PAVEL SMRŽ, Ph.D.**
SUPERVISOR



BRNO 2017

Zadání diplomové práce/17871/2016/xmatou21

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

# Zadání diplomové práce

Řešitel: **Matoušek Martin, Bc.**
Obor: Počítačové sítě a komunikace
Téma: **Paralelní a distribuované zpracování rozsáhlých textových dat**
**Parallel and Distributed Processing of Large Textual Data**
Kategorie: Web

Pokyny:

1. Prostudujte základní přístupy k paralelnímu a distribuovanému zpracování velkých objemů dat a existující řešení pro sledování výkonnosti takových systémů.
2. Seznamte se s dostupnými nástroji pro analýzu textu v přirozeném jazyce.
3. Na základě získaných poznatků navrhněte a realizujte systém, který pro zpracování rozsáhlých textových dat optimálně využije výpočetní infrastrukturu dostupnou naší skupině.
4. Vyhodnoťte realizované řešení a porovnejte zvolený přístup z hlediska efektivity a integrovatelnosti.
5. Vytvořte stručný plakát prezentující práci, její cíle a výsledky.

Literatura:

- dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Funkční prototyp

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

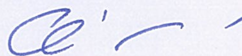
Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrž Pavel, doc. RNDr., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.



doc. Dr. Ing. Jan Černocký
*vedoucí ústavu*

## Abstrakt

Diplomová práce se zabývá plánováním úloh a přidělováním prostředků v paralelním a distribuovaném prostředí. Popisuje návrh a implementaci aplikace pro spouštění zpracování dat s optimálním využitím dostupných zdrojů.



## Abstract

This master thesis deals with task scheduling and allocation of resources in parallel and distributed enviroment. Thesis subscribes design and implementation of application for executeing of data processing with optimal resources usage.



## Klíčová slova

distribuovaný systém, paralelní systém, zpracování textových dat, plánování úloh, Apache Mesos

## Keywords

distributed system, parallel system, processing of textual data, task scheduling, Apache Mesos


## Citace

MATOUŠEK, Martin. *Paralelní a distribuované zpracování rozsáhlých textových dat*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. RNDr. Pavel Smrž, Ph.D.

# Paralelní a distribuované zpracování rozsáhlých textových dat

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. RNDr. Pavla Smrže, Ph.D. Další informace mi poskytl Ing. Jaroslav Dytrych. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.......................
Martin Matoušek
24. května 2017

## Poděkování

Zde bych chtěl poděkovat vedoucímu práce Doc. RNDr. Pavlu Smržovi, Ph.D za připomínky a konzultace, které mi poskytl. Poděkování patří také panu Ing. Jiřímu Jarošovi, Ph.D za pomoc se superpočítačem Salomon.

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project "IT4Innovations National Supercomputing Center - LM2015070"

# Obsah


**1 Úvod** **3**

**2 Distribuované systémy** **4**
2.1 Architektury distribuovaných systémů . . . 4
2.1.1 Grid . . . 5
2.1.2 Cluster . . . 5
2.2 Plánovací proces . . . 5
2.2.1 Úloha . . . 5
2.2.2 Zdroj . . . 6
2.2.3 Závislosti mezi úlohami . . . 7
2.2.4 Rozvrhovací problémy . . . 7
2.3 Plánovač . . . 7
2.3.1 Architektura plánovačů . . . 7
2.3.2 Statické plánování . . . 8
2.3.3 Dynamické plánování . . . 8
2.4 Algoritmy plánování . . . 9
2.4.1 Algoritmy založené na základních heuristikách . . . 9
2.4.2 Algoritmy založené na lokálním prohledávání . . . 10
2.4.3 Genetické algoritmy . . . 11
2.5 Plánování zdrojů . . . 11
2.5.1 Monolitický plánovač . . . 12
2.5.2 Dvouúrovňový plánovač . . . 12
2.5.3 Plánovač sdílených stavů . . . 12
2.6 Používané systémy . . . 12
2.6.1 Condor . . . 12
2.6.2 Maui . . . 13
2.6.3 Portable Batch System (PBS) . . . 13
2.6.4 YARN . . . 13
2.6.5 SLURM . . . 13
2.6.6 Borg . . . 13
2.6.7 Apache Mesos . . . 14

**3 Přístupy ke zpracování dat** **15**
3.1 Sériové zpracování . . . 15
3.2 Zřetězené zpracování . . . 15
3.3 MapReduce . . . 16

**4 Apache Mesos** **17**

4.1 Architektura . . . 17
4.1.1 Master . . . 17
4.1.2 Slave . . . 18
4.1.3 Framework . . . 18
4.2 Existující frameworky . . . 19
4.2.1 Marathon . . . 19
4.2.2 Chronos . . . 19
4.3 Zdroje . . . 19
4.4 Izolace zdrojů . . . 20
4.5 Komunikace . . . 20
4.6 Přidělování zdrojů . . . 21

**5 Návrh aplikace** **23**
5.1 Omezení aktuálního způsobu zpracování . . . 23
5.2 Požadavky na aplikaci . . . 23
5.3 Návrh aplikace . . . 24
5.4 Vyhodnocení statistik . . . 24
5.5 Objektový návrh . . . 25

**6 Implementace** **26**
6.1 Použité technologie . . . 26
6.1.1 Java . . . 26
6.1.2 SQLite . . . 26
6.1.3 Python . . . 26
6.1.4 SYSSTAT . . . 27
6.1.5 Webové technologie . . . 27
6.2 Třída Main . . . 27
6.3 Třída DB . . . 28
6.4 Třídy MpiExecutor a StdExecutor . . . 28
6.5 Třídy pro fronty a úlohy . . . 29
6.6 Třída MyCheduler . . . 29
6.7 Třída SimulatedAnnealing . . . 32
6.8 Vyhodnocení naměřených hodnot . . . 33

**7 Testování a vyhodnocení systému** **34**
7.1 Zpracování . . . 35
7.1.1 Zpracování na clusteru Salomon . . . 35
7.1.2 Zpracování na serveru Knot . . . 35
7.1.3 Výsledky zpracování . . . 36
7.1.4 Naměřené statistiky . . . 37

**8 Závěr** **39**

**Literatura** **40**

**A Obsah přiloženého paměťového média** **43**

**B Plakát** **44**

# Kapitola 1

# Úvod

Systémy pro distribuci a paralelizaci úloh jsou v dnešní době velmi rozšířeným řešením. Zpracovávají náročné úlohy z oblasti vědy a výzkumu, na které by běžné počítače neměly dostatečný výkon. Dále se používají pro zajištění velké dostupnosti poskytovaných služeb pro zákazníky. Pro firmy je využití takovýchto systémů velice výhodné, protože nemusí spravovat vlastní infrastrukturu a mají k dispozici moderní technologie a velký výpočetní výkon.

Představme si situaci, ve které potřebujeme zpracovat velké množství souborů. K dispozici máme programy, které postupně zpracovávají soubory, potřebujeme je spustit na distribuovaném systému tak, abychom co nejlépe využili dostupný hardware. Pokud spuštěná zátěž nebude odpovídat výkonu systému, dojde k plýtvání zdrojů a financí.

Cílem této diplomové práce je nastudovat techniky plánování úloh v distribuovaných a paralelních systémech a vytvořit aplikaci, která bude sloužit pro optimální plánování jednotlivých kroků při zpracování dat.

Diplomová práce je členěna do dvou částí: teoretické a praktické. V teoretické části v kapitole 2 je popsán způsob plánování úloh v distribuovaných systémech a algoritmy pro plánování, v závěru kapitoly jsou představeny zástupci z distribuovaných systémů. V kapitole 4 je představen systém Apache Mesos, který jsem zvolil jako základ pro vytvoření aplikace. Současný způsob zpracování a návrh nové aplikace je rozebrán v kapitole 5. Kapitola 6 se zabývá implementací a vnitřní funkčností aplikace. Testování a naměřené výsledky jsou obsaženy v kapitole 7. Závěrečná kapitola 8 se zabývá celkovým zhodnocením diplomové práce.

# Kapitola 2

# Distribuované systémy

Distribuovaný systém je takový systém, kde se jednotlivé součásti nacházejí v síti a společně komunikují pomocí zasílání zpráv. V této kapitole se podíváme na několik základních architektur distribuovaných systémů. V dnešní době se využívají v moderních aplikacích, jako jsou webové vyhledávače, zpracování textu, online hry a finanční systémy.

Sdílení zdrojů je hlavním důvodem pro výstavbu a využívání distribuovaných systémů. Nemusí se jednat pouze o výpočetní výkon, ale také o další komponenty (například disky, tiskárny, databáze a další). Distribuované systémy zahrnují mnoho technologických inovací z posledních let a porozumění těmto technologiím je zásadní pro moderní práci na počítači [5].

## 2.1 Architektury distribuovaných systémů

Distribuované systémy jsou postaveny na existujících sítích a operačních systémech. Obsahují skupinu autonomních počítačů spojených přes počítačovou síť a distribuovaný Middleware. Middleware umožňuje počítačům koordinovat své činnosti a sdílet zdroje systému tak, že uživatelé vnímají systém jako jediný celek. Middleware je tedy most mezi aplikací a hardwarovou částí systému. Distribuovaný systém může být propojen libovolnou strukturou:

- Plné propojení - propojení každý s každým. Problém takového systému je při přidávání dalšího uzlu do systému, což vede k velkému nárůstu propojovacích linek. Náklady na komunikaci mezi uzly jsou velmi malé a zpráva odeslaná z jednoho uzlu jde do druhého přímo. Plně propojené systémy jsou spolehlivé, protože i přes selhání několika počítačů můžou ostatní stále komunikovat.

- Částečné propojení - přímé propojení existuje jen mezi několika uzly. Propojení může mít strukturu podobnou například stromu, hvězdě, sběrnici, nebo kruhu. Některé tradiční distribuované systémy, jako je server/klient, využívají hvězdu jako topologii sítě. Problém nastane v případě výpadku centrálního uzlu, kdy okolní uzly nemohou komunikovat. Dalším problémem je delší doba komunikace mezi uzly, pokud informace prochází přes více uzlů.

### 2.1.1 Grid

Grid je označení pro systém využívající distribuované heterogenní výpočetní zdroje propojené LAN nebo WAN sítí. Počítače připojené do sítě mohou být prostorově odděleny i na velké vzdálenosti.

Výpočty prováděné na gridu sdílejí přístup k velkému množství výpočetního výkonu a úložného prostoru. Velkou předností gridu je dynamika zdrojů, které mohou být volně přidávány nebo odebírány podle potřeby. Informační služba gridu uchovává informace o zdrojích, jako je identifikátor, dostupnost, kapacita, typ a aktuální zátěž. Uživatelé vkládají úlohy do plánovače a ten je přiřadí dostupnému zdroji podle aktuální zátěže. Stroje zapojené do gridu se mohou lišit v mnoha ohledech, jako je počet procesorových jednotek, výpočetní výkon jednotek a vnitřní politika plánování [17].

Aplikace spouštěné na gridu řeší problémy v oblastech výzkumu (aplikovaná fyzika, jaderná fyzika, geologie, seismologie, mechanika a matematika). V dnešní době se čím dál víc objevují i komerční aplikace z oblastí zpracováni velkých dat, jako jsou databáze, data mining a diagnostika.

### 2.1.2 Cluster

Cluster je systém složený z procesorů propojených rychlou sítí. Jsou vytvářeny pro zvýšení výpočetního výkonu, který vysoce převyšuje výkon samostatného počítače. Slouží pro paralelní výpočty nebo se používají pro zajištění vysoké dostupnosti poskytovaných služeb [33].

V clusterech se využívá Master/Slave architektura plánování. Hlavní uzel sbírá úlohy, které přiděluje podřízeným uzlům. Nejčastěji mají uzly clusteru stejné parametry, jako je CPU, paměť a síťová propustnost. Cílem clusteru je zvýšení výkonu s centrální správou zdrojů [25].

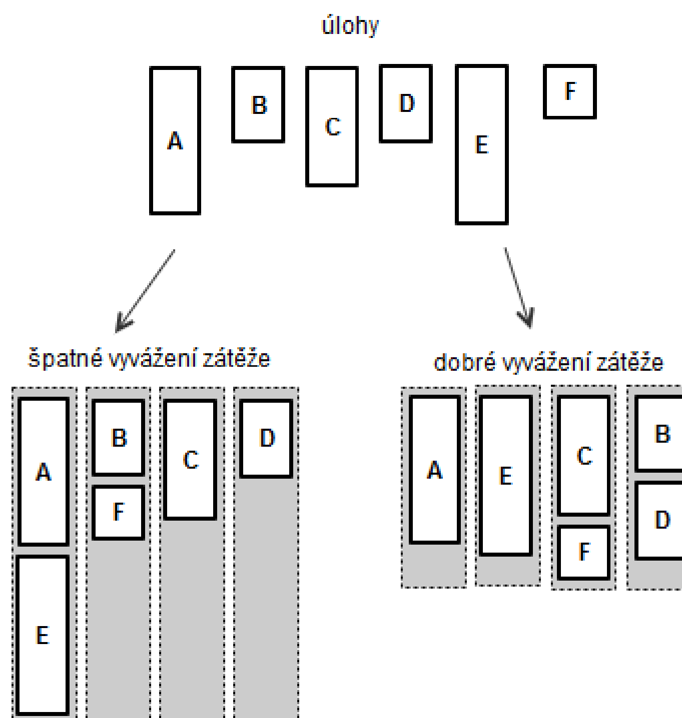
## 2.2 Plánovací proces

Plánování je proces pro určení pořadí úloh, ve kterém budou úlohy zpracovány. V distribuovaných systémech je plánování daleko komplikovanější než v jednoprocesorových systémech. Výběr vhodného plánovacího algoritmu je důležitý pro maximální propustnost. Pro velmi malý počet úloh je možné nalézt optimální plán přiřazení zdrojů, ale s přibývajícími úlohami roste počet možností, v jakém pořadí je spustit. Pro problémy tohoto typu není možné nalézt řešení v optimálním čase, proto se používají aproximace, které naleznou suboptimální řešení v rozumném čase. Existují plánovací algoritmy založené na heuristikách, které najdou přibližné řešení problému [17].

Plánování neřeší pouze problém přiřazení jednotlivých úloh na zdroje ve správném pořadí, ale musí také řešit vyrovnání zátěže. Tím zaručíme vytížení dostupných zdrojů na maximum, které může vést ke snížení hodnoty Makespan. Makespan udává celkovou dobu zpracování a často bývá hlavním kritériem při porovnání plánovacích algoritmů. Zdroje s optimálním vyrovnáním zátěže pracují efektivněji, jak je znázorněno na obrázku 2.1.

### 2.2.1 Úloha

Je základní entita, která je plánována na zdroje. V systému by mělo být minimálně tolik úloh, jako je výpočetních jednotek. Čím více bude úloh, tím získáme větší flexibilitu pro plánování. Výpočet každé úlohy musí dostatečně kompenzovat režii s řízením úloh. Úlohy




Obrázek 2.1: Příklad vyvážení zátěže

mají statické vlastnosti, které získávají při vytvoření. Naopak dynamické vlastnosti získají po přiřazení na stroj. Seznam vlastností úloh [7]:

- Processing time (doba zpracování) - čas zpracování úlohy na konkrétním stroji
- Release date (termín dostupnosti) - čas, kdy je možné nejdřív začít se zpracováním
- Due date (termín dokončení) - čas, kdy musí být úloha dokončena
- Weight (váha úlohy) - používá se jako priorita
- Start time (začátek zpracování) - čas, kdy se začne provádět úloha na daném stroji
- Completion time (konec zpracování) - čas, kdy končí zpracování úlohy na daném stroji

### 2.2.2 Zdroj

Zdroj je základní jednotka, na kterou se přiřazují úlohy ke zpracování. Každý zdroj má omezení v podobě počtu CPU, paměti, rychlosti a zátěže. Rychlost definuje dobu zpracování úlohy na daném stroji a zátěž určuje aktuální vytížení zdroje. Tyto informace o každém zdroji jsou poskytnuty plánovači, který je využívá při plánování úloh. Máme několik různých skupin zdrojů [7]:

- Jeden samostatný stroj
- Identické paralelní stroje se stejnou rychlostí
- Paralelní stroje s různou rychlostí doby zpracování úloh

- Nezávislé paralelní stroje s různou rychlostí

### 2.2.3 Závislosti mezi úlohami

Existují dva typy úloh. První z nich jsou nezávislé úlohy, které nemají mezi sebou žádný vztah a mohou být vykonány v libovolném pořadí se stejným výsledkem.

Druhou skupinou jsou závislé úkoly, které mají značný dopad na možnosti plánování. Je nutné naplánovat tyto úlohy ve správném pořadí pro získání správného výsledku. Závislosti mezi úlohami jsou zobrazovány pomocí orientovaného acyklického grafu nazývaného DAG (direct acyclic graph). Uzly grafu představují jednotlivé úlohy a orientované hrany slouží pro znázornění závislostí mezi úlohami. Jedná se o primární model používaný pro reprezentaci závislých úloh při plánování [25].

### 2.2.4 Rozvrhovací problémy

Rozvrhování se dnes využívá v mnoha oborech. Například při zpracování výroby, plánování pracovních směn, plánování v prostředí gridu a při návrzích univerzitních rozvrhů. K popisu rozvrhovacích problémů se používá Grahamova klasifikace. Klasifikační schéma $\alpha|\beta|\gamma$ je možné použít pro reprezentování problémů plánování v distribuovaných prostředích a výrobních procesech. $\alpha$ představuje klasifikaci zdrojů a způsob alokace, $\beta$ znamená klasifikaci úloh a omezující podmínky a $\gamma$ jsou optimalizační kritéria.

Multi-operační problémy můžeme rozdělit na tři kategorie podle způsobu provádění jednotlivých operací [7]:

- Open shop - plánování rozdílných úloh pomocí plánovače
- Flow shop - úloha prováděna na všech strojích ve stejném pořadí (výrobní linka)
- Job shop - úloha prováděna na všech strojích v daném pořadí

Podle těchto definic může být plánování nezávislých úloh spojeno s problémem Open shop a plánování pro závislé úlohy přirovnáno k Job shop problému. Flow shop problém odpovídá plánování cyklických částí [25].

## 2.3 Plánovač

Plánovač hraje důležitou roli v distribuovaných a paralelních systémech. Přistupuje ke zdrojům a k úlohám a snaží se nalézt optimální kombinaci jejich přiřazení. Snaží se přiřadit relativně velký počet úloh na relativně malý počet zdrojů. Toto rozhodování může záviset na vlastnostech úloh a dostupnosti zdrojů. Naplánování aplikace zahrnuje následující kroky: nalezení zdrojů, výběr zdrojů, vygenerování plánu a vykonání úloh. V případě dynamického plánování mohou procesy způsobit dodatečné kroky, jako je přeplánování nebo přizpůsobení [25].

### 2.3.1 Architektura plánovačů

Rozlišujeme tři plánovače: centralizovaný, hierarchický a distribuovaný.

- Centralizovaný plánovač - Centrální stroj nebo uzel řídí ostatní zdroje a plánuje úlohy na všechny okolní uzly, které patří do skupiny. Toto paradigma je často použito v případě, kdy mají zdroje podobnou charakteristiku. Úlohy jsou nejdříve vloženy do cen-

trálního plánovače, který rozhodne o přidělení úkolu. Pokud úkol nemůže být nikam přidělen, zůstává ve frontě [19].

Centralizovaný model není dobře škálovatelný při zvětšujícím se počtu zdrojů. Plánovač se může stát v některých situacích slabým místem. Například pokud dojde k chybě spojení mezi plánovačem a okolními uzly. Naopak výhodou je schopnost vytvořit efektivní plán přidělení úloh, protože centrální plánovač má přehled o všech dostupných zdrojích [35].

- Distribuovaný plánovač - Neobsahuje centrální plánovač pro udržování všech úkolů. Je založen na distribuovaném plánování, které provádějí lokální plánovače. Skupina plánovačů mezi sebou komunikuje a společně přiřazuje úlohy výpočetním uzlům. Plánovače komunikují přímou cestou, kde jsou všichni propojeni navzájem, nebo přes společnou množinu úloh, ze které si vybírají úlohy [19].
- Hierarchický plánovač - Plánovací proces je sdílen mezi několika úrovněmi plánovačů. Nejvyšší plánovač přiděluje skupinu úkolů nižším plánovačům, kteří dále rozdělují úlohy na jednotlivé zdroje. V porovnání s centrálním plánovačem je hierarchický model lepší ve škálování s rostoucím počtem zdrojů [35].

### 2.3.2 Statické plánování

Statické plánování znamená, že dopředu známe všechny úlohy a k jejich plánování dochází před startem aplikace. Jednotlivé úkoly jsou spojeny do bloků a přiřazeny výpočetní jednotce. Velikost bloků je upravena tak, aby každá výpočetní jednotka pracovala přibližně stejnou dobu. Aplikace často využívají statické plánování, pokud jsou výpočetní zdroje dostupné po celou dobu výpočtu. Někdy je označováno jako offline plánování [23].

### 2.3.3 Dynamické plánování

Je využíváno, když je doba zpracování úloh nepředvídatelná, nebo když mají jednotlivé výpočetní jednotky rozdílné výkony. Nejčastější způsob dynamického plánování je společná fronta úloh, ke které mají přístup všechny výpočetní jednotky. Když jednotka dokončí přidělený úkol, vyzvedne si další z fronty. Rychlejší jednotky budou přistupovat k frontě častěji než ty pomalejší. Někdy označováno jako online plánování [23].

Dynamické algoritmy plánování jsou schopny zvládnout jakékoli narušení způsobené změnou prostředí. Tyto změny mohou být rozděleny do tří skupin. První skupinou je změna množiny úloh, která může způsobit nekonzistenci v plánování například novou příchozí úlohou. Druhou skupinou jsou změny dostupných zdrojů. Nekonzistenci způsobuje jejich nedostupnost nebo zpomalení. Poslední skupinou jsou časové změny, které zapříčiní například zpožděné dokončení úlohy.

Při vyrovnání s nekonzistencí máme různé možnosti. Jednou z možností je provést nové plánování a zahrnout změny. Tato varianta je časově náročná a málo používaná. Dále můžeme použít lokální opravy v předchozím plánu a touto možností nepřijdeme o pracně vytvořený plán [18].

Volba plánovací strategie pro daný problém není vždy jednoduchá. Při statickém plánování vzniká malá režie během paralelního výpočtu a měla by být použita nejčastěji. Pokud nám do systému přicházejí úlohy náhodně, musíme použít plánování dynamické [23].

## 2.4 Algoritmy plánování

### 2.4.1 Algoritmy založené na základních heuristikách

#### Min-Min a Max-Min

Algoritmus začíná s množinou nepřiřazených úloh MT. V první fázi je pro všechny úlohy z MT vypočítána *minimální očekávaná doba výpočtu*. Ve druhé fázi je úkol s nejmenší hodnotou *minimální očekávané doby výpočtu* vybrán z množiny MT a přiřazen na odpovídající zdroj. Poté je úkol odebrán z množiny MT a postup se opakuje, dokud nejsou naplánovány všechny úlohy [9].

```
while (J != Null) // J je množina úloh
for each job J_i in J
  for each machine M_i in M // M je množina strojů
    calculate C_ij = E_ij + R_j
    // C_ij,E_ij, R_j reprezentují čas dokončení,
    // čas výpočtu a čas přípravy na úlohu na příslušném stroji
  end for
end for
for each job J_i in J
  Find the minimum completion time and the machine that obtains it
end for
Search the job J_u having minimum completion time among all unassigned jobs
Allocate J_u to machine M_v that has resulted in obtaining
        minimum completion time of J_u.
Delete job J_u from the job set J: J = J - J_u
Update the ready time of machine M_v as: R_v = C_uv
End While
```

Obrázek 2.2: Algoritmus Min-Min (převzato z [9])

Algoritmus Max-Min je stejný jako Min-Min, ale dochází k výpočtu a výběru úlohy s největší dobou zpracování.

#### Opportunistic Load Balancing (OLB)

Plánuje každou úlohu v náhodném pořadí na další očekávaný dostupný zdroj bez ohledu na dobu zpracování úlohy na daném stroji. Záměrem OLB je držet všechny stroje maximálně vytížené. Výhoda algoritmu je v jeho jednoduchosti, ale nebere v úvahu očekávanou dobu zpracování úlohy [24].

#### Minimum Execution Time (MET)

Přiřazuje každou úlohu v náhodném pořadí na stroj, kde bude zpracována v nejkratším čase bez ohledu na dostupnost zdroje. MET se snaží přiřadit každý úkol na nejvhodnější stroj. To může způsobit nevyvážení zátěže a neefektivní využití zdrojů [24].

### Minimum Completion Time (MCT)

Přiřazuje každou úlohu v náhodném pořadí na stroj, kde bude mít úloha nejmenší minimální čas dokončení. MCT kombinuje výhody OLB a MET a vyhýbá se slabinám obou algoritmů [24].

### First Come First Served (FCFS)

Algoritmus je založen na frontě, ze které jsou postupně odebírány úlohy. Jedná se o velice jednoduchý a snadno implementovatelný algoritmus. Umožňuje prioritní plánování pomocí více váhových front. Je velmi rozšířen v HPC[1] systémech a nejčastěji je doplněn technologií Backfill.

Backfill je plánovací optimalizace, která poskytuje plánovači lepší využití dostupných zdrojů. Jsou nalezeny úlohy mimo pořadí zpracování a s nimi jsou zaplněny volné zdroje. Tím dojde k lepšímu využití zdrojů a urychlení doby čekání ve frontě na dostupné prostředky.

### Dominant Resource Fairness (DRF)

DRF je založen na algoritmu min-max fairness pro plánování vícenásobných zdrojů, které min-max nezvládne. Využívá se v clusterech, kde je potřeba přidělovat několik typů zdrojů, jako je CPU, paměť, porty, disk a další.

Příklad rozdělování zdrojů [6]:

Pro ukázku máme dostupné tyto zdroje: 8 CPU a 10 GB paměti. První uživatel spustí úlohy, které spotřebují 1 CPU a 3 GB paměti, druhý uživatel spotřebuje 3 CPU a 1 GB paměti. Po dokončení rozdělování zdrojů bude mít první uživatel přidělené 2 CPU a 6 GB paměti a druhý uživatel dostane 6 CPU a 2 GB paměti.

### 2.4.2 Algoritmy založené na lokálním prohledávání

Lokální prohledávání je významnou oblastí pro řešení problému. Do této kategorie řadíme algoritmy Hill climbing (horolezecký algoritmus) a Simulated annealing (simulované žíhání). Jsou založeny na vyhledávacích algoritmech, které prohledávají stavový prostor. Základní myšlenka je opakovaně generovat a vyhodnocovat kandidáty řešení.

### Hill climbing

Horolezecký algoritmus je interaktivní technika, která začíná s náhodným řešením ve stavovém prostoru, poté se pokouší nalézt jeho optimální řešení postupnou modifikací jednotlivých elementů. Pokud modifikace přinese lepší řešení, tak nahradí řešení předchozí. V opačném případě je horší modifikace zrušena [34].

Tento proces se opakuje tak dlouho, dokud nejsou možné další změny konfigurace. Pokud toto nastane, je s velkou pravděpodobností nalezeno lokální minimum [30].

### Simulated annealing

Algoritmus je založen na lokálním prohledávání pro vyřešení problémů. Velkou výhodou algoritmu je vyhnutí se uváznutí v lokálním minimu při prohledávání okolních stavů. Pro-

[1] High-performance computing - typ počítačového clusteru

hledávání začíná náhodným výběrem prvků z počáteční množiny. Dále jsou vygenerovány všechna možná řešení, ze kterých se vybere nové řešení a s ním se pokračuje dále.

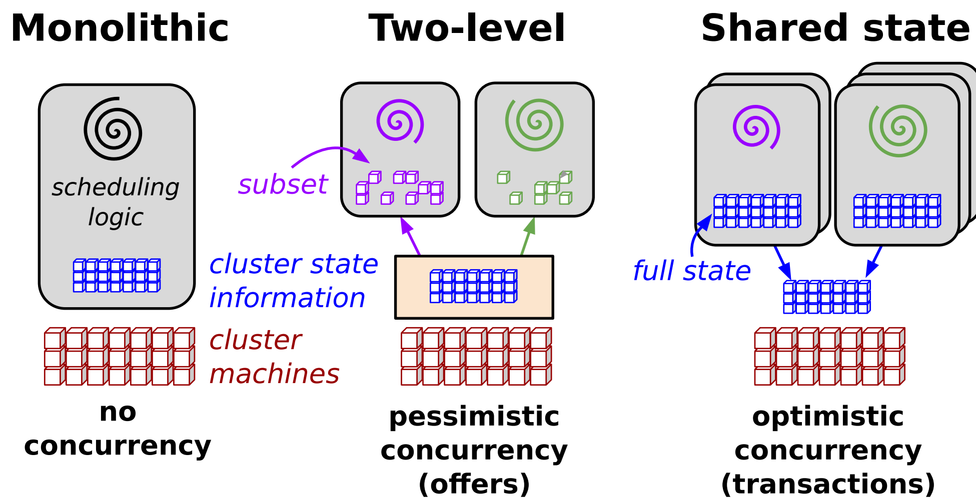
### 2.4.3 Genetické algoritmy

Genetické algoritmy připomínají proces přírodního vývoje. Generují náhodnou počáteční populaci možných kandidátů řešení. Každé řešení reprezentuje vektor indexů, který se nazývá chromozóm. Následně je na chromozómy aplikována operace křížení a mutace. Algoritmus má omezený počet kroků, a to redukuje pravděpodobnost uváznutí v lokálním minimu [34].

## 2.5 Plánování zdrojů

Velké výpočetní clustery jsou drahé, a proto je nutné jejich maximální využití. Efektivita a využití můžou být zvýšeny spuštěním rozdílných úloh na stejném stroji. Máme úlohy závislé na CPU a paměti, krátkodobé, nebo dlouhodobé. Poskytnutí volnosti pro typy úloh přináší lepší využití hardwaru, ale vytváří plánovací problémy (přiřazení úloh na stroje). Zatímco cluster zůstává stejný, musí plánovač vykonávat stále složitější plánování a může se stát slabým prvkem v systému [28].

Plánovače clusteru mají několik úkolů. Musí efektivně rozdělit zdroje mezi zadanou práci a zajistit oddělení jednotlivých úloh, aby se vzájemně neomezovaly. Dnes převládají dvě architektury: monolitický a dvouúrovňový plánovač.




Obrázek 2.3: Schéma plánovacích architektur (převzato z [28])

### 2.5.1 Monolitický plánovač

Plánovač běží v jedné instanci na libovolném počítači z clusteru a rozhoduje o spuštění úloh. Celé plánování je provedeno stejným plánovačem a všechny úlohy jsou plánovány stejnou logikou. Jedná se o jednoduché chování, které je snadno implementovatelné, proto je monolitický plánovač velice rozšířený. Problém nastává v případě, že potřebujeme pro některé úlohy jinou plánovací politiku. Toho je možno dosáhnout pomocí různých kódů, jedná se ale o velmi náročný krok [28].

Při velkém počtu zadaných úloh může dojít k přetížení plánovače, tím se zpomalí plánování čekajících úloh.

Monolitický plánovač je nejčastěji používán v HPC systémech. Do této kategorie patří systémy, jako je Condor, YARN, Maui, PBS, SLURM a další.

### 2.5.2 Dvouúrovňový plánovač

Tento typ plánovače odděluje plánování úloh od přidělování zdrojů. Využívá centrálního koordinátora, který získává informace o dostupných zdrojích od uzlů. Ze získaných informací sestaví nabídky zdrojů a rozhodne, který program nabídku obdrží. Daný program na základě vlastní plánovací politiky rozhodne o využití nabízených zdrojů. Tím je přesunuto plánování úloh na vyšší vrstvu a centrální koordinátor se zabývá pouze volnými zdroji. Informace o celém clusteru má jen centrální uzel a ostatní aplikace mají informace jenom o vlastních zdrojích [28]. Do této kategorie patří například Apache Mesos.

### 2.5.3 Plánovač sdílených stavů

Skládá se z několika plánovačů s vlastní rozhodovací politikou, které si udržují aktuální kopii stavu celého clusteru. Neobsahuje centrální prvek, který by určoval pořadí plánovačů, a získání zdrojů probíhá soubojem mezi všemi plánovači. Vyhraje plánovač, který dříve přistoupí ke zdroji a pomocí atomické operace aktualizuje stav clusteru. Tedy vždy alespoň jeden z plánovačů alokuje zdroje [28]. Do této skupiny patří například systém Omega[2], Nomad[3] a Apollo[4].

## 2.6 Používané systémy

### 2.6.1 Condor

Condor je jeden z prvních distribuovaných systémů pro plánování dávkového zpracování. Je založen na mechanizmu front, plánovací politice, prioritním schématu, monitorování zdrojů a správě zdrojů. Uživatelé spouštějí úlohy a Condor je ukládá do front a určí kdy, jaká úloha a kde bude spuštěna podle plánovací politiky [31].

Centrální uzel udržuje informace o ostatních uzlech v databázi s jejich charakteristikami. Plánování probíhá pomocí shody mezi plánovanou úlohou a dostupnými zdroji. Pro úlohu je nalezena nejlepší shoda se záznamem v databázi dostupných zdrojů, které jsou následně přiděleny úloze, ta je poté spuštěna.

[2] Více informací na: https://research.google.com/pubs/pub41684.html

[3] Více informací na: https://www.nomadproject.io/

[4] Více informací na: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-boutin_0.pdf

### 2.6.2 Maui

Produkt s názvem Maui [13] je vytvořen firmou Cluster Resources Inc. pro správu řízení zátěže a plánování úloh na clusteru. Je primárně navržen pro dávkové zpracování pomocí front, ale umožňuje spouštět i úlohy v přesně zadaný čas pomocí kalendáře. Úlohy jsou vkládány do front a při dostatku zdrojů jsou spouštěny.

Obsahuje správu zdrojů Torque[5], který je integrován do plánovače a poskytuje několik možností při plánování, jako je priorita úloh, rovnoměrné přidělování zdrojů a Backfill. Cluster se skládá z hlavního uzlu a pracovních uzlů.

Uživatelům jsou poskytnuty příkazy pro příkazovou řádku, kterými mohou komunikovat s hlavním uzlem. Mají možnost vkládat úlohy do fronty a kontrovat jejich stav. Vložení úlohy se provádí pomocí příkazu `qsub` a specifikací náročnosti úlohy, jako je čas běhu, CPU a paměť.

### 2.6.3 Portable Batch System (PBS)

PBS zajišťuje řízení zátěže pro clustery nebo HPC systémy. Byl navržen NASA, protože předchozí distribuované systémy byly nedostatečné, a tak navrhla moderní systém pro svoje potřeby [15]. Plánování je prováděno algoritmem FCFS s rozšířením Backfill, který je vložen pomocí modulu. Tento algoritmus je možné nahradit vlastním plánovačem.

Uživatel komunikuje pomocí příkazů, které se odesílají hlavnímu procesu. Ten se stará o fronty s úlohami a provádí jejich plánování na dostupné zdroje a uživatelům poskytuje informace o jejich úlohách.

### 2.6.4 YARN

YARN [1] je plánovač, který přišel s Hadoop[6] verze 2.0. Jedná se o správu zdrojů pro Hadoop ekosystém. Hlavní uzel je nazýván ResourceManager a spravuje informace o zdrojích a spuštěných aplikacích. Komunikuje s pracovními uzly NodeManager, které se starají o kontejnery, monitorování použitých zdrojů a posílání zpráv hlavnímu uzlu.

RM obsahuje plánovač, který je zodpovědný za alokování prostředků pro běžící aplikace a spouští naplánované aplikace. Druhou komponentou RM je ApplicationsManager, který přijímá úlohy od klientů a vyjednává zdroje potřebné pro spuštění úlohy.

### 2.6.5 SLURM

SLURM [14] je open source správa zdrojů pro počítače a je složen z několika komponent. Klíčovým prvkem je `slurmctld daemon`, který je centrálním prvkem a řídí celý systém. Na uzlech je spuštěn proces `slurmd daemon`, jenž se stará o správu přiděleného uzlu. Jeho úkolem je spouštění a monitorování úloh a informování o stavu uzlu.

Uživatelé přistupují k plánovači podobně jako u systému PBS. Pomocí příkazů vkládají úlohy do fronty, ze které jsou pomocí plánovače vybírány a spouštěny na dostupných uzlech.

### 2.6.6 Borg

Systém vytvořený společností Google pro správu clusteru, který dokáže řídit stovky tisíc úloh. Spouští libovolné úlohy, do kterých patří například dlouhodobé služby jako Gmail a Google Docs [32].

[5] Více informací na: http://www.adaptivecomputing.com/products/open-source/torque/

[6] Více informací na: https://hadoop.apache.org/

Jednotlivé stroje jsou rozděleny do buněk, které mohou obsahovat tisíce heterogenních strojů. Centrální kontrolér nazývaný Borgmaster provádí plánování a komunikuje s procesy označovanými jako Borglet, které běží uvnitř buněk na každém stroji. Borglety se starají o spuštění, zastavení, popřípadě restartování přidělených úloh. Dále spravují zdroje uzlu a dostupné zdroje odesílají na vyžádání uzlu Borgmaster.

### 2.6.7 Apache Mesos

Dalším distribuovaným systémem je Apache Mesos vyvinutý v AMPLab, UC Berkeley v roce 2011. Při jeho nasazení na cluster nabízí vývojářům a uživatelům široké možnosti při využívání clusteru. Poskytuje různé typy zpracování, mezi které patří dávkové zpracování, interaktivní analýzy, MPI zpracování, sdílení dat, webové aplikace a různé typy frameworků. Jde o silný nástroj, který je velmi flexibilní a odolný proti chybám [16].

Mesos využívá sdílení zdrojů, čímž je dosažena velká flexibilita a propustnost oproti statickému rozdělování zdrojů mezi aplikace.

K rozšíření systému Mesos na clustery pomohla výborná komunita. Díky modulární architektuře máme možnost pro mnoho úprav tak, aby vyhovoval našim požadavkům. Mesos dnes využívají mnohé společnosti, mezi které patří například Twitter a Airbnb [6].

# Kapitola 3

# Přístupy ke zpracování dat

V této kapitole se zaměříme na možné způsoby, jak zpracovávat data. Rozebereme jejich průběh a porovnáme vlastnosti při jejich použití. U každého přístupu ke zpracování dat bude uveden příklad, kde se používá.

## 3.1 Sériové zpracování

Při sériovém zpracování máme skupinu samostatných programů, kde každý program provádí vždy jeden krok zpracování. Vstupem a výstupem je soubor, a proto dochází k většímu zatížení disků. Každý spuštěný program má vždy dostupná data, jejichž dostupnost je omezena pouze rychlostí disku.

Pokud program nebo soubor obsahuje chybu a dojde k selhání programu, přijdeme pouze o jeden krok zpracování. Zrychlení zpracování dat můžeme dosáhnout na víceprocesorových systémech spuštěním většího počtu instancí programu.

### Projekt COW

Příkladem je projekt COW na Svobodné univerzitě v Berlíně, který vytváří webové korpusy. Využívají program `texrex` pro čištění textu od HTML značek a skriptů, odstranění duplicitních dokumentů a klasifikování dokumentů. Pro další operace používají dostupné nástroje třetích stran. Data zpracovávají na clusteru univerzity, který je založen na systému SLURM [26].

Díky systému SLURM mohou jednotlivé kroky paralelizovat na více uzlech a pomocí spouštěcího skriptu jednoduše vyměnit aktuální programy za jiné [27].

## 3.2 Zřetězené zpracování

Jedná se o propojení existujících programů postupně za sebe a data prochází napříč programy. Protože se data ihned předávají dalšímu programu, je odstraněn mezikrok se zápisem výstupu do souboru a jeho následným čtením dalším programem.

Komplikace mohou nastat v případě, že některý z programů nebo souborů obsahuje chybu. Tím ztratíme celé zpracování daného souboru, které budeme muset zpracovávat znovu, protože nemáme mezivýsledky. Dalším problémem je rychlost výstupu programů, při kterém musí program čekat na data od pomalého předchůdce. I když zřetězené zpracování spustíme paralelně s maximálním počtem souběžných zpracování, nemusíme využít celkový výpočetní výkon, protože některé programy mohou čekat na data.

### Stanford CoreNLP

CoreNLP je nástroj pro zřetězené zpracování implementovaný v Javě. Vznikl spojením existujících programů do jednoho celku, který nabízí několik operací pro zpracování textu jako je čištění, nalezení duplicit, určení slovního druhu a další. Řízení zajišťuje anotátor, který vybírá operace k provedení [21].

## 3.3 MapReduce

MapReduce je programovací model pro zpracování velkých dat. Hlavní myšlenkou je rozdělit zpracování na několik menších celků a odděleně je zpracovat. Obsahuje dvě fáze: fázi map a fázi reduce. Základní strukturou jsou páry obsahující klíč a hodnotu, které jsou vstupem a výstupem obou fází.

Operace map vezme vstupní data a provede programátorem definovanou operaci. Výstupem jsou pracovní páry, které jsou operací reduce sloučeny podle stejných klíčů.

### DKPro C4Corpus

Zástupcem MapReduce je balíček nástrojů DKPro C4Corpus pro zpracovávání CommonCrawl. Jedná se o framework pro Hadoop platformu. Při vytváření byl kladen důraz na velkou škálovatelnost, díky které umožňuje spouštět zpracování současně na více než 2000 uzlech. Obsahuje základní operace pro zpracování textu jako je čištění, nalezení duplicitních dat, detekce jazyka a další [10].

# Kapitola 4

# Apache Mesos

Systém Mesos bude základem pro novou aplikaci. Proto se v této kapitole podrobněji podíváme na jeho vnitřní strukturu a možnosti, které poskytuje při vytváření aplikací pro zpracování dat.

## 4.1 Architektura

Mesos umožnuje spouštět libovolné aplikace tak, aby je nebylo potřeba upravovat pro běh v distribuovaném prostředí. Může se jednat o klasické programy nebo frameworky, jako jsou Hadoop, Spark, Kafka a další. Centrálním prvkem je uzel Master, který řídí celý cluster. Master komunikuje s ostatními uzly nazývanými Slave.

Charakteristika Mesos architektury [6]:

- Škálovatelnost – cca 50 000 uzlů
- Izolace zdrojů – pomocí Linux / Docker[1] kontejneru
- Efektivita – přidělování zdrojů (např. CPU, paměť)
- Rozhraní pro monitorování – webové UI o aktuálním stavu clusteru

### 4.1.1 Master

Master je zodpovědný za zprostředkování komunikace mezi pracovními uzly Slave a frameworkem. V systému může být spuštěno více uzlů typu Master, ale aktivní může být pouze jeden. Při selhání aktivního Master uzlu dojde pomocí aplikace Zookeeper[2] k aktivování čekajícího Mastera. Nový Master je vybrán pomocí distribuovaného protokolu [16].

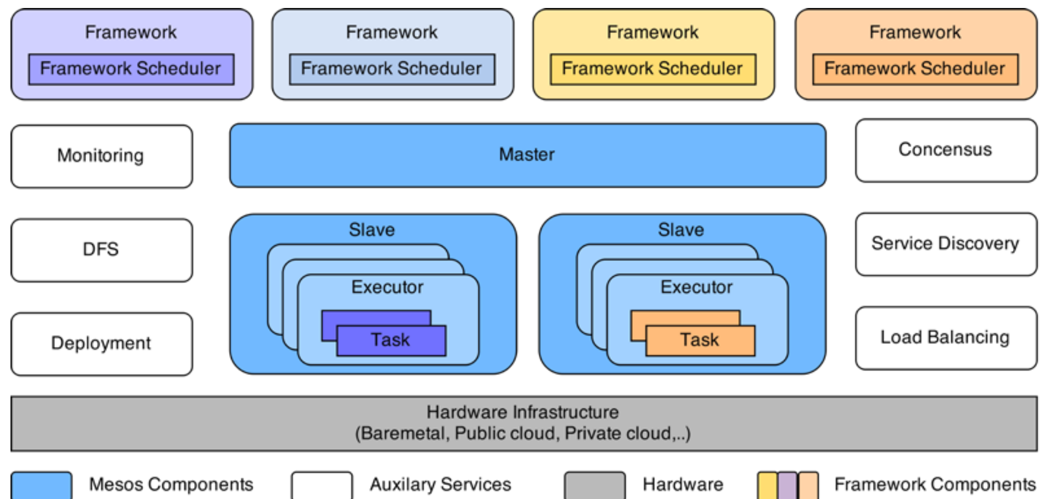
Master slouží pouze pro získání dostupných zdrojů od Slave uzlů. Následně vyhodnotí, kterému frameworku budou zdroje nabízeny. Protože se Master stará pouze o zprostředkování nabídek, je efektivní i při velkém počtu uzlů a není zatěžován náročnými výpočty.

Master je mozkem celého clusteru a má tyto vlastnosti [8]:

- Centrální důvěryhodný zdroj pro běžící úlohy
- Spravedlivé rozdělení zdrojů mezi frameworky

[1] Více informací na: `https://www.docker.com/`
[2] Více informací na: `https://zookeeper.apache.org/`

- Poskytuje webové rozhraní
- Nabízí vysokou dostupnost služeb a efektivní využití zdrojů




Obrázek 4.1: Mesos architektura (převzato z [16])

### 4.1.2 Slave

Slave je spuštěn na každém uzlu, který je zařazen do zpracování. Stará se o spravování zdrojů a informuje o nich uzel Master.

Všechny zdroje jsou zabaleny a odeslány uzlu Master, který rozhodne o jejich přidělení a odešle je konkrétnímu frameworku.

Slave je zodpovědný za provedení úloh přidělených frameworkem. Je nutné, aby Slave poskytl správnou izolaci prostředků pro úlohy, které běží současně. Izolační mechanizmy zajití, že úlohy dostanou přesně ty zdroje, o které požádaly [16].

### 4.1.3 Framework

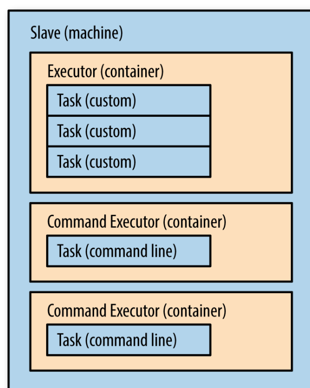
Framework je aplikace běžící nad uzlem Master, která řídí plánování úloh a jejich spuštění. Dříve byla komunikace mezi frameworkem a hlavním uzlem založena na knihovně implementované v jazyce C++, a proto nebylo možné použít jiné programovací jazyky. Dnes využívá protokol HTTP a dají se použít i jiné programovací jazyky, jako je Java, Scala, Python nebo Go. Framework obsahuje dva prvky: exekutor a plánovač [6].

Plánovač přijímá od uzlu Master seznam nabídek s dostupnými zdroji pomocí modulu SchedulerDriver, který zajišťuje jejich komunikaci. Podle vlastní rozhodovací politiky plánovač rozhodne, jestli nabízené zdroje využije. V případě přijetí nabídky je vytvořen exekutor na uzlu Slave, kterému jsou následně předány informace o množství spotřebovaných zdrojů a informace o úlohách.

Exekutor běží na uzlu Slave a stará se o spouštění přidělených úloh a izolaci potřebných zdrojů. Pomocí vnitřní komunikace mezi exekutorem a Slave uzlem jsou předávány infor-

mace o aktualizaci stavu jednotlivých úloh. Aktualizace stavu jsou z uzlu Slave odeslány modulem ExecutorDriver do uzlu Master, který tyto informace předá frameworku.

Když exekutor přijme úlohu od plánovače, provede implementovaný výpočet. Pro spuštění externího programu slouží objekt CommandExecutor, který provede zadaný příkaz [6].




Obrázek 4.2: Vztah mezi exekutorem a Slave uzlem (převzato z [8])

## 4.2 Existující frameworky

### 4.2.1 Marathon

Marathon je framework pro spouštění aplikací s dlouhou dobou běhu, jako jsou například webové služby. Poskytuje funkce pro jednoduché spuštění aplikací v prostředí clusteru a zaručuje jejich dostupnost. Může být použit jako výchozí framework, který bude spouštět ostatní frameworky a kontrolovat jejich stavy. Podporuje i běžné spuštění programů z příkazové řádky. Široké nastavení pro jednotlivé úlohy dovoluje přesně určit, kde a s jakými parametry se má daná aplikace spustit [8].

### 4.2.2 Chronos

Nahrazuje starší verzi Cron. Chronos je distribuovaný plánovač pro aplikace s dlouhou dobou běhu, ale i pro opakující se aplikace jako například zálohování. Pomocí ISO8601 notace podporuje spuštění aplikace v přesně zadaný čas. Velkou výhodou je spouštění úloh, které jsou závislé na dokončení předchozí úlohy. Díky tomu je možné vytvořit řetězové zpracování. Obsahuje vlastní webové rozhraní pro monitorování spuštěných úloh [8].

## 4.3 Zdroje

Pro přidělování zdrojů se používá alokační modul, ten provádí rozhodování, kterému frameworku a v jakém množství budou nabídnuty dostupné zdroje. Výchozí modul používá algoritmus DRF, který spravedlivě přidělí zdroje všem aktivním frameworkům.

Úlohy a exekutoři konzumují zdroje při vykonávání práce. Každý uzel obsahuje různé zdroje, které mohou být využity. Základní zdroje obsahují počet CPU jader, velikost paměti, síťové porty a velikost disku. Mimo základní zdroje máme také atributy, které uchovávají informace o uzlech, ale tyto atributy slouží pouze pro identifikace uzlů a nemohou být

spotřebovány úlohami. Základní zdroje jsou popsány pomocí skalárních hodnot, rozsahu, množiny nebo textového řetězce.

Přehled standardních zdrojů [8]:

- cpus – vyjadřuje počet dostupných CPU jader. Úlohy mohou využívat i zlomek jádra, protože Mesos používá sdílení CPU, a tak je možné alokovat například pouze 50 % jádra. To znamená, že při alokaci 1,5 cpus bude procesu přiděleno 1,5 sekund strojového času každou sekundu. Když budou procesy dva, každý dostane 750 ms strojového času, nebo dostane jeden proces 1 s a druhý pouze 500 ms strojového času.
- mem – označuje, kolik paměti bude potřeba. Na rozdíl od cpus nemůžeme paměť přetížit, když se tak stane, dojde k ukončení úlohy. Vybrání správné kapacity paměti je kritickou úlohou. Málo přidělené paměti ukončí proces, ale naopak příliš mnoho přidělené paměti bude neefektivní.
- disk – udává množství úložného místa
- ports – značí síťové porty, které budou používány

Mesos umožňuje vlastní nastavení dostupných zdrojů pro uzly Slave při jejich spuštění parametrem `--resources` např.:

```
--resources='cpus:30;mem:12288;disk:921;ports:[2100-2900];names:{a,b}'
```

## 4.4 Izolace zdrojů

Pokud na jednom uzlu spouštíme více úloh různých uživatelů, nesmějí se navzájem ovlivňovat. K tomuto účelu slouží kontejnery, které působí jako obal poskytující izolaci. Použití kontejnerů omezuje množství zdrojů, jež proces může využít. Mesos umožnuje použít různé implementace kontejnerů pomocí modulů. Standardně poskytuje dva typy izolace: izolace na základě procesů, která je na POSIX systémech; druhou možností je použití Cgroups pro izolaci na systému Linux [16].

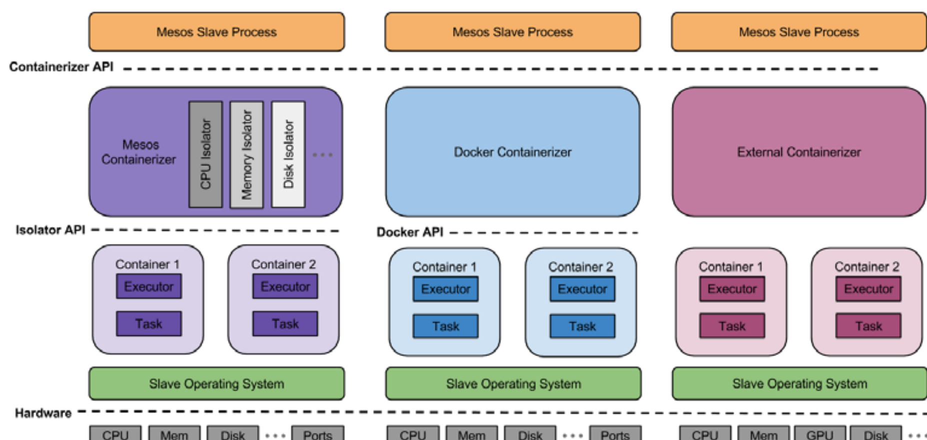
Pokud uživatel potřebuje specifické chování, může použít jiné řešení v podobě dnes rozšířeného kontejneru Docker, nebo vytvořit vlastní řešení.

## 4.5 Komunikace

Pro interní komunikaci mezi uzly využívá Mesos protokol HTTP. Při implementaci používá knihovnu `libprocess`, která poskytuje asynchronní komunikaci mezi procesy. Komunikace probíhá pomocí následujících API [16]:

- Scheduler API – komunikace mezi frameworkem a uzlem Master
- Executor API – komunikace pro exekutor a uzel Slave
- Internal API – komunikace pro uzly Master a Slave
- Operator API – pro webové rozhraní

Pokud chce nějaká část komunikovat, sestaví HTTP POST požadavek s informacemi a ten je odeslán adresátovi.



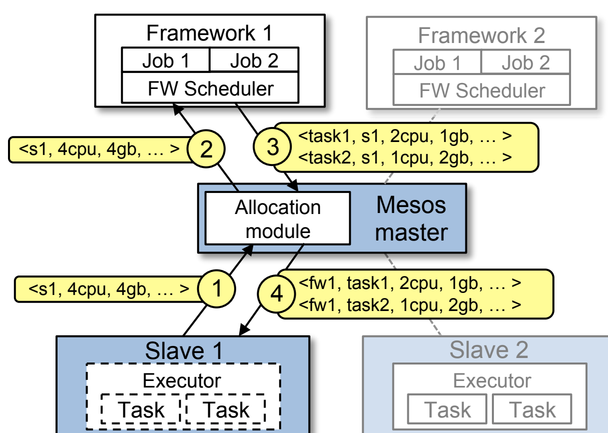

Obrázek 4.3: Mesos Slave s rozdílnými mechanizmy izolace (převzato z [16])

## 4.6 Přidělování zdrojů

Mesos je založen na dvouúrovňovém plánovači, který se stará o přidělování zdrojů mezi frameworky. Na první úrovni získává uzel Master informace o volných zdrojích ze všech uzlů typu Slave a seskupí je do nabídky. Podle rozhodovací politiky je vybrán framework, kterému bude nabídka zaslána.

Druhou úroveň značí framework, který nabídku zdrojů od uzlu Master přijme podle vnitřní politiky frameworku. Když je nabídka přijata, framework odešle informace o úlohách a jejich potřebných zdrojích zpět na uzel Master, který je dále předá na konkrétní uzel, a exekutor vykoná zadanou práci. Po dokončení práce jsou zdroje uvolněny a opět nabízeny k dalšímu použití.

Příklad přidělení zdrojů [6]:




Obrázek 4.4: Příklad přidělení zdrojů (převzato z [12])

1. Slave 1 předá uzlu Master informace o volných zdrojích (4 CPU, 4 GB paměti). Master pomocí alokačního modulu rozhodne o přidělení všech zdrojů frameworku 1.
2. Master odešle nabídku frameworku 1 o dostupných zdrojích.
3. Plánovač frameworku 1 přijme zdroje a odešle informace o dvou úlohách a jejich potřebných zdrojích (2 CPU, 1 GB paměti a 1 CPU, 2 GB paměti)
4. Master odešle úlohy na uzel Slave 1 a ten alokuje zdroje pro exekutor, který následně spustí úlohy. 1 CPU a 1 GB paměti jsou stále volné a budou znovu zaslány uzlu Master.

# Kapitola 5

# Návrh aplikace

V této kapitole představím současný způsob zpracování textu skupinou Knot, návrh celkové aplikace a její strukturu. Podrobně popíšu jednotlivé komponenty, které budou sloužit pro plánování úloh, sledování systému a ke komunikaci mezi jednotkami.

## 5.1 Omezení aktuálního způsobu zpracování

Výzkumná skupina Knot vytváří sémanticky anotované texty z webových dat. Při zpracování textu provédějí tyto kroky: převod do vertikálního formátu, odstranění duplicit, dále morfologické, syntaktické a sémantické značkování. Jednotlivé kroky jsou aktuálně spouštěny odděleně. Tím vniká několik problémů, které zpomalují zpracování. Prvním problémem je čekání na dokončení zpracování všech souborů v rámci jednoho kroku, než se pokračuje dalším krokem. S každým dalším krokem roste doba čekání.

Druhým problémem je náročnost na zdroje při zpracování. Pokud je některý z programů na zpracování například náročný na paměť, může nastat situace, při které dojde k zaplnění paměti, a přesto nebudou plně využita všechna jádra procesoru.

Každý krok je spouštěn centrálně s pevným nastavením. Problém může nastat, pokud máme různorodý hardware pro servery. V tomto případě můžeme spustit zpracování podle konfigurace nejslabšího serveru, nebo manuálně spustit zpracování pro jednotlivé servery.

Informace sledované při zpracování jsou omezeny na čas strávený na procesoru. Ostatní statistiky nejsou sledovány.

## 5.2 Požadavky na aplikaci

Cílem aplikace je optimálně využít dostupné zdroje pro zpracování dat a výsledkem by mělo být zkrácení doby zpracování. Vytvořený program bude pro prostředí superpočítače Salomon a fakultní systém Knot, který se skládá z oddělených serverů.

Program by neměl být zaměřen pouze na zpracovávání textu, ale měl by být, pokud možno univerzální, s možností spouštět libovolné programy. Díky tomu by měl umožnit zpracování i jiných dat, jako je klasifikace obrázků, zpracování videa a vyhledávání v textu. Důležitou vlastností bude jednoduchá výměna programů pro zpracování a provedení jejich porovnání mezi sebou. Konfigurace programu přes vstupní soubor bude určovat posloupnost jednotlivých kroků, které budou použity při zpracování.

Aplikace naplánuje zpracování různých paralelních procesů s optimálním využitím zdrojů (paměti, procesoru, disku) a podle aktuálního využití zdrojů budou spuštěny další úlohy.

Tak vznikne možnost spuštění jednotlivých kroků pro testování nových programů nebo celkového zpracování všemi kroky. Z výsledků bude možné vyhodnotit náročnost programů a určit programy pro optimalizaci nebo zobrazit získané statistiky o zpracování.

## 5.3 Návrh aplikace

Systém musí fungovat na dvou primárních systémech. Prvním je cluster Salomon, který má společný datový prostor. Druhým prostředím je grid Knot, který se skládá z několika serverů s různou konfigurací. Každý server v gridu má lokální datové úložiště. Jako centrální logiku jsem zvolil Apache Mesos popsaný v kapitole 4, a to z několika důvodů.

Většina systémů počítá při plánování s časem dokončení úlohy. V našem případě je tato hodnota proměnlivá, protože doba zpracování souboru závisí na jeho vnitřní složitosti. Mesos umožňuje přidělit pouze část procesorového jádra oproti ostatním systémům, které pracují s nejmenší alokační jednotkou jedno jádro. Pokud budeme spouštět aplikaci, která využije pouze 1,5 jádra, nebudeme muset alokovat dvě celá jádra a 0,5 jádra může být použito pro další přidělení.

Aplikace bude navržena jako framework pro Mesos. V teoretické části 2.6 jsou popsány nejznámější frameworky, ale žádný z nich neumí pracovat s MPI, a proto jsem zvolil tvorbu vlastního frameworku.

Plánování bude probíhat vždy pro každý uzel zvlášť, protože Mesos vytváří nabídky zdrojů pro každý uzel. Tím odstraníme nedostatky, které by vznikly při celkovém plánování všech úloh. V situaci, kdy by bylo potřeba naplánovat několik tisíc úloh, by docházelo k pomalému plánování. Další zpomalení by nastalo v případě, kdy bude uzel zatížen prací, protože ve většině případů nemáme možnost spustit plánovač na samostatném stroji. V našem případě bude nejčastěji plánovač spuštěn na některém z pracovních uzlů.

Pro vyhodnocení úloh, které je možné spustit na dostupných zdrojích pro daný uzel, použijeme algoritmus simulovaného žíhání. Úlohy budou vybírány z množiny zástupců pro každý krok, tím zajistíme rychlost algoritmu a kombinaci rozdílně náročných úloh.

O každém zpracovaném souboru budou sbírány statistiky a po dokončení výpočtu dojde k uložení statistik do databáze. Později dojde k vyhodnocení vlastností všech souborů a celkového zpracování.

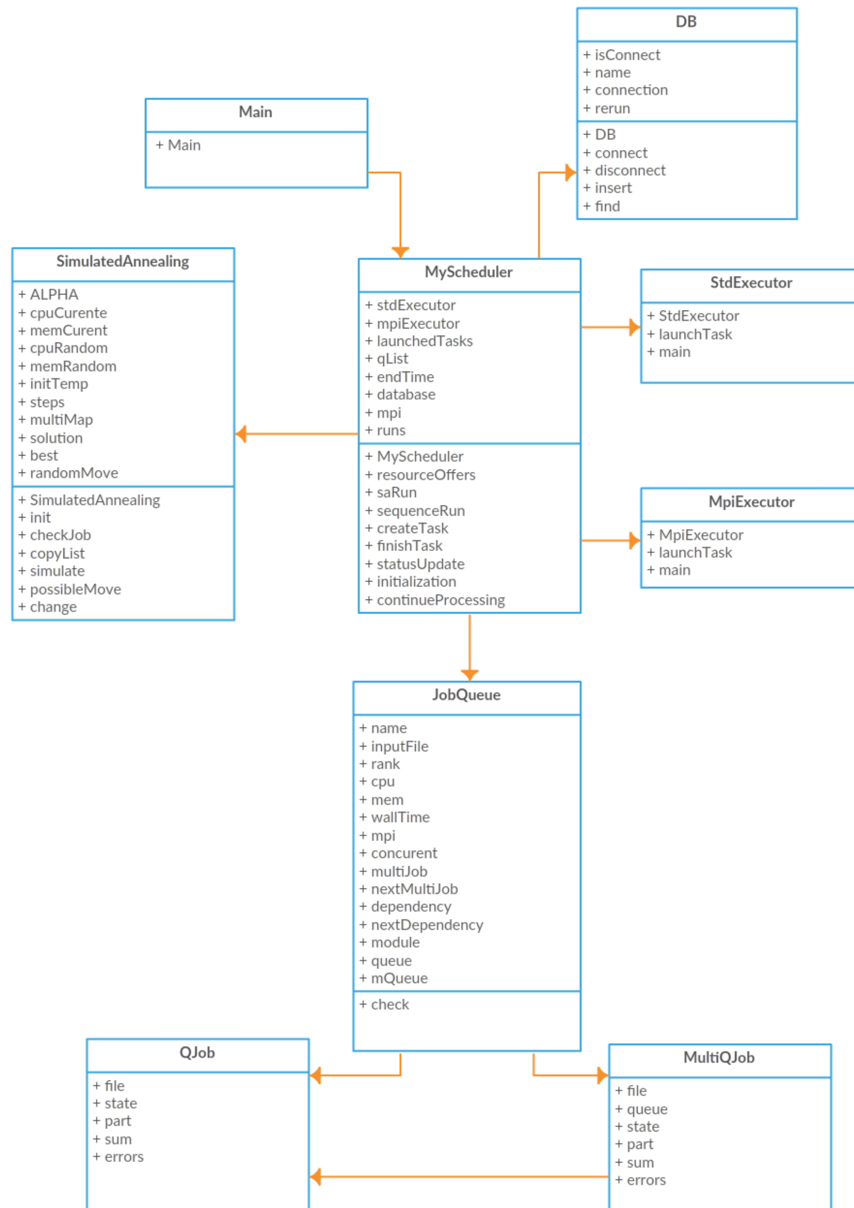
## 5.4 Vyhodnocení statistik

Aplikace bude sledovat několik vlastností spuštěných úloh. Jedná se o dobu zpracování, přidělenou paměť a procesorovou náročnost. Tyto statistiky budou sloužit pro vyhodnocení spuštěných programů a nalezení slabých částí systému, pro které by se mohly vytvořit optimalizace. Dalším přínosem statistik bude porovnání nově připravených komponent zapojených do zpracování.

Důležitou vlastností je zkoumání závislostí na velikosti vstupních souborů a nalezení odchylek doby zpracování nebo neobvyklého nároku na paměť. Pokud se takový soubor vyskytne, může být určen pro prozkoumání a nalezení příčiny nárůstu doby zpracování. Ze získaných hodnot bude vytvořena webová stránka s přehledy pro jednotlivé soubory.

## 5.5 Objektový návrh

Třídy aplikace a jejich vztahy jsou znázorněny na diagramu tříd, který je zobrazen na obrázku 5.1. V diagramu nejsou uvedeny části, které nemají zásadní vliv na chod aplikace. Na zobrazeném diagramu je vidět, že je jádrem aplikace třída `MyScheduler`, která provádí plánovaní a odesílá úlohy ke zpracování na exekutory. Jednotlivé třídy jsem se snažil navrhnout tak, aby odpovídaly reálnému světu. Třída `DB` se bude starat o komunikaci s databází. Třídy `MpiExecuter` a `StdExecuter` představují agenty na pracovních uzlech a vykonávají přidělené úlohy. Třída `JobQueue` představuje fronty pro vkládání úloh `QJob` a `MultiQJob`.




Obrázek 5.1: Zjednodušený diagram tříd

# Kapitola 6

# Implementace

Tato kapitola se zabývá implementací celé aplikace a pomocných skriptů. Budou popsány jednotlivé třídy a metody použité v aplikaci. Na začátku kapitoly představím technologie a knihovny, které jsou využívány v aplikaci.

## 6.1 Použité technologie

### 6.1.1 Java

Java je kompilovaný i interpretovaný jazyk. Zdrojový kód je přeložen do binárních instrukcí podobně jako běžný strojový kód mikroprocesoru, ale je vykonán bezpečně ve virtuálním prostředí. Každá aplikace napsaná v Javě může být spuštěna na libovolné platformě, která obsahuje JRE prostředí. Proto není nutné vytvářet různé verze programu pro rozdílné platformy [22].

Zvolení Javy hlavním programovacím jazykem zaručí, že bude celá aplikace kompatibilní s většinou systémů. Díky tomu zůstane aplikace univerzální a nebude potřeba nová kompilace pro různé systémy. V porovnání se skriptovacími jazyky je Java daleko rychlejší a také se více hodí pro rozsáhlé aplikace. Na druhou stranu je pomalejší než jazyk C++, ale tato vlastnost je vyvážena snadnou přenositelností.

### 6.1.2 SQLite

SQLite je knihovna pro vytvoření a manipulaci s databází. Na rozdíl od běžné databáze nemusí být přímo spuštěna na serveru, ale je možné databázi vytvořit do souboru nebo do fyzické paměti. SQLite dokáže zabalit celou databázi do jediného souboru, který obsahuje informace o struktuře s aktuálními daty v tabulkách. Formát tohoto souboru je přenositelný na libovolné platformy a může být použit na libovolném hardwaru [4].

Vytvoření databáze v jediném souboru je zde klíčové. Díky této vlastnosti bude zajištěna přenositelnost aplikace i databáze a nebude nutné hledat vhodný stroj, na kterém by musela běžet klasická databáze.

### 6.1.3 Python

Jedná se o interpretovaný a objektově orientovaný jazyk. První verzi vytvořil Guido van Russom už v roce 1991. Jako interpretovaný jazyk je přenositelný na různé platformy a má mnoho knihoven třetích stran [20].

Jako skriptovací jazyk je vhodný pro práci s textem, a proto jsem použil Python pro vytvoření webové stránky s grafy a naměřenými statistikami.

### 6.1.4 SYSSTAT

SYSSTAT je utilita pro analýzu a monitorování stavu operačního systému a úloh na něm běžících. Obsahuje několik monitorovacích nástrojů (sar, sads, sadf, iostat, pidstat, ...) pro globální analýzu systému. Dokáže monitorovat velký počet operací a prostředků, jako jsou CPU statistiky, paměť, virtuální paměť, přerušení, síťové operace, souborový systém, odkládání stránek a další. Tyto informace umožňuje generovat v několika formátech [3].

Z dostupných nástrojů bude použit pidstat, který bude monitorovat jednotlivé moduly a zaznamenávat jejich procesorovou a paměťovou náročnost.

### 6.1.5 Webové technologie

Aplikace získává jednotlivé statistiky, ze kterých se vytvářejí přehledy a grafy. Proto jsem zvolil jako vizualizační formu webovou stránku, která bude obsahovat všechna shromážděná data.

#### HTML a CSS

HyperText Markup Language (HTML) a kaskádové styly (CSS) jsou základní stavební prvky web designu a jejich současné požití vytváří moderní webovou stránku [11].

#### Highcharts

Highcharts je knihovna napsána v JavaScriptu pro kreslení grafů, vznikla v roce 2009. Při vytváření mysleli vývojáři i na mobilní zařízení a vytvořili podporu pro dotyková gesta [2].

## 6.2 Třída Main

Aplikace se spouští z třídy `Main`, ve které se nejprve zkontrolují vstupní parametry. Parametr `-t` určuje použití kontroly času při spuštění úloh. Tento parametr je vhodný pro časově omezené zdroje, jako je například přidělení uzlů na superpočítači Salomon. Pokud parametr není zadán, aplikace předpokládá neomezený čas pro zpracování a časy zadané v konfiguračním souboru pro jednotlivé úlohy jsou ignorovány. Parametr `-f` označuje konfigurační soubor a parametr `-c` značí pokračování v rozdělané práci. Posledním parametrem je `-a`, který obsahuje adresu a port pro uzel Master. Po zkontrolování parametrů dojde k vytvoření standardního a MPI exekutoru metodou `ExecutorInfo.newBuilder()`. Příkazem `setName()` je nastaven jejich název a příkazem `setValue()` je přiřazena cesta k jejich třídám.

Funkcí `FrameworkInfo.newBuilder()` je vytvořen framework, který se následně registruje pro přijímání nabídek se zdroji od uzlu Master. Dále dojde k vytvoření instance `MyScheduler`, které jsou předány informace o zadaných vstupních parametrech, připojené databázi a reference na exekutory. Řízení komunikace zajišťuje třída `MesosSchedulerDriver`, jež je spuštěna příkazem `driver.run()`. Vstupním parametrem při vytváření je třída `MyScheduler`, která je jádrem pro plánování a spouštění úloh.

```
ExecutorInfo executorSTD = ExecutorInfo.newBuilder()
  .setExecutorId(ExecutorID.newBuilder().setValue("std"))
  .setCommand(CommandInfo.newBuilder()
  .setValue("java -cp " + JAR_PATH
          + " cz.vutbr.fit.mesosframework.executors.StdExecutor"))
  .setName("Executor for standard app")
  .build();
```

Obrázek 6.1: Ukázka vytváření exekutoru

## 6.3 Třída DB

Vytvoření databáze probíhá ve třídě `DB`, která zajišťuje připojení a vytvoření databáze do souboru funkcí `DriverManager.getConnection("jdbc:sqlite:"+name)`. Pokud se jedná o první spuštění aplikace, je vytvořena nová databáze a tabulka `completed` uchovávající záznamy o dokončených úlohách. Pro komunikaci s databází slouží dvě metody. Metoda `insert` pro vkládání nových záznamů a metoda `find`, která vyhledává záznamy při kontrole dokončených úloh pro obnovení předchozího stavu při pokračování zpracování.

Do databáze jsou ukládány informace o frontě, ze které je soubor vybrán, procesorové a paměťové nároky, název zpracovaného souboru, celkový čas zpracování, celkový čas strávený na procesoru a stav zpracování.

| completed |
|---|
| ID integer <<PK>> |
| queue text |
| file text |
| memory real |
| cpu real |
| time text |
| status integer |
| timecpu text |

Obrázek 6.2: Struktura tabulky pro statistiky

## 6.4 Třídy MpiExecutor a StdExecutor

Pro spouštění modulů slouží dvě třídy `StdExecutor` a `MpiExecutor`. Exekutoři jsou spuštěni na všech uzlech Slave, ze kterých se dále spouštějí jednotlivé úlohy. Každý exekutor dostává jako parametry název vstupního a výstupního souboru nebo složky, název fronty, ze které je soubor vybrán, a modul, který má být spuštěn. V případě MPI jsou data doplněna o názvy uzlů s počtem procesů, na kterých se má MPI spustit, ze kterých je vytvořen konfigurační soubor pro `mpiexec`. Když exekutor dostane úkol, vytvoří nové vlákno a oznámí jeho spuštění funkcí `sendStatusUpdate` s hodnotou `TASK_RUNNING` a spustí modul funkcí `exec`. Poté se spuštěná úloha začne monitorovat. Prvním krokem je zjištění `pid` spuštěného modulu, podle kterého najdeme všechny podprocesy, o nichž budeme získávat

statistiky. Pomocí aplikace `pidstat` monitorujeme periodicky každé 2 sekundy všechny nalezené podprocesy a sledujeme alokovanou paměť a procesorovou náročnost. Čas strávený na procesoru je monitorován příkazem `time`, kde informace získáme po dokončení úlohy. Po dokončení modulu jsou statistiky odeslány zpět třídě `MySchedule` spolu s aktualizací stavu `TASK_FINISHED` funkcí `sendStatusUpdate`. V případě selhání úkolu jsou odeslány nulové statistiky a oznámení o nedokončení úkolu pro nové naplánování.

## 6.5 Třídy pro fronty a úlohy

### Třída JobQueue

Fronta je implementována ve třídě `JobQueue`. Uchovává informace z konfiguračního souboru (CPU, paměť, čas, ...), které se používají při alokaci zdrojů na jednotlivých uzlech. Dále obsahuje seznam klasických úloh `QJob`, nebo seznam skupinových úloh `MultiQJob`.

### Třída QJob

Každá úloha je definována třídou `QJob`, jež uchovává informace o dané úloze. Obsahuje název souboru, který bude zpracovávat, informace o stavu (ready, wait, run), identifikátor v rámci skupiny a počet chybných pokusů o spuštění. Pokud je překročen limit pro chybné spuštění, je úloha zaznamenána v databázi jako nedokončená. Po označení nebude dále zahrnuta do plánování.

### Třída MultiQJob

Druhou třídou pro úlohu je `MultiQJob`, která slouží pro uchování skupiny úloh při zadání parametru `PART`. Má stejné vlastnosti jako třída `QJob` a navíc obsahuje seznam úloh, které se mají zpracovat současně.

## 6.6 Třída MyCheduler

V této třídě je implementováno načítání konfiguračního souboru, obnovení při znovuspuštění a komunikace mezi exekutory s třídou `SchedulerDriver`. Třída uchovává informace o všech frontách, připojení k databázi a časové značce pro vyhodnocení dostatku času pro zpracování. Ve třídě jsou implementovány tyto metody:

**Registered** je volána, když dojde k úspěšné registraci frameworku.

**ResourceOffer** je periodicky volána, přijde-li nabídka zdrojů od uzlu Master. Každá nabídka obsahuje pouze informace o jednom uzlu. Po přijetí nabídky dojde k naplánovaní spuštění úloh. Jako první jsou naplánovány úlohy MPI, protože jsou spouštěny na více uzlech. Pomocí cyklu získáme celkové zdroje ze všech nabídek, čímž získáme celkové dostupné zdroje. Poté prohledáme fronty s parametrem MPI a zkontrolujeme možnost spustit některou z úloh. Pokud najdeme úlohu, která není spuštěna, zkontrolujeme dostatek času na provedení úkolu (pokud je zadám parametr -t), dále zkontrolujeme, jestli není zadáno postupné zpracování. Pokud jsou všechny podmínky splněny, dojde ke kontrole potřebných zdrojů. Při dostatku zdrojů dojde k vytvoření úlohy `taskInfo`, pro kterou nastavíme potřebné zdroje CPU a potřebnou paměť programu. Dále vytvoříme data, která budou

předána s úlohou do exekutoru. Data obsahují název vstupního a výstupního souboru, název fronty, modul a seznam, na kterých uzlech se bude MPI spouštět. Vytvořená úloha je odeslána příkazem `launchTasks`, který předá informace exekutorovi, ten následně spustí zadaný modul. Nyní jsou potvrzeny nabízené zdroje.

Když není možné spustit MPI úlohu, přejdeme k naplánování klasických úloh funkcí `saRun`.

Po zpracování všech nabídek zdrojů a spuštění maximální možné zátěže jsou nevyužité zdroje odmítnuty, poté dojde ke kontrole běhu úloh. Pokud tato kontrola zjistí 10krát za sebou, že není spuštěna žádná úloha, dojde k ukončení programu. Tento stav nastane v případě, že není dostatek času na spuštění dalších úloh. Když dojde k ukončení, ale přesto nejsou fronty prázdné, je nastavena návratová hodnota frameworku na jedničku, která značí ukončení pro nedostatek času na zpracování.

**SaRun** je metoda pro plánování klasických úloh, pro které je použit algoritmus simulovaného žíhání implementovaný ve třídě `SimulatedAnnealing`. Nabídky zdrojů jsou postupně procházeny a plánování probíhá pro každou nabídku zvlášť. Nejprve dojde ke kontrole podmínek pro spuštění, jako v případě MPI. Kontroluje se čas potřebný ke běhu, postupné zpracování a nároky na zpracování. Pokud máme úlohy, které můžeme spustit, dojde k vybrání 24 zástupců z každé fronty úloh a vytvoření seznamu, který obsahuje vybrané úlohy a jejich náročnost. Seznam a nabídnuté zdroje se předají algoritmu pro plánování ve třídě `SimulatedAnnealing`, který určí úlohy pro spuštění. Funkcí `simulate` získáme seznam úloh a z nich vytvoříme funkcí `createTask` úlohy, které spustíme příkazem `launchTasks`.

**SequenceRun** je alternativa pro simulované žíhání. Jedná se o plánování, kde probíhá výběr cyklickým procházením front a výběrem vždy jedné úlohy.

**StatusUpdate** je volána při změně stavu úlohy jako start, dokončení, běh a selhání. Při změně stavu na `TASK_RUNNING` dojde pouze k zaznamenání stavu. V případě dokončení úlohy jsou přijata data od exekutoru se statistikami, které jsou zpracovány a uloženy do databáze metodou `insert`. Samotná úloha je potom převedena z aktuální fronty do fronty následujícího kroku a označena pro plánování metodou `finishedTask`.

Posledním možným stavem je selhání úlohy. V tomto případě je úloha znovu označena pro zpracování a je zvýšeno počítadlo selhání úlohy. Když úloha dosáhne čtyř selhání, předpokládáme, že se jedná o špatný soubor, nebo chybu v programu, jenž provádí zpracování. Do databáze je uložen záznam o nedokončeném zpracování a úloha se při dalším plánování neuvažuje.

Po každé dokončené úloze je kontrolováno, jestli zbývají další úlohy ke zpracování. Pokud už žádné úlohy nezbývají, dojde k ukončení aplikace.

**CreateTask** je funkce pro vytvoření objektu `task`, který předává informace exekutorovi. Objekt obsahuje název, přidělené ID, identifikaci uzlu, na kterém bude spuštěn a potřebné zdroje. Dále jsou připojena data o frontě a souboru, který se bude zpracovávat.

**FinishedTask** je metoda sloužící pro přesun úloh mezi frontami při dokončeném zpracování. Nejprve se zjistí, jestli má být dále zpracována, nebo zda už prošla posledním krokem. Jestliže se jednalo o poslední krok, je pouze odstraněna z fronty. V opačném případě je nalezena fronta, do které bude zařazena, a vloží se buďto přímo do fronty, nebo v případě skupinového zpracování do seznamu pro skupinové zpracování.

```
TaskInfo task = TaskInfo
  .newBuilder()
  .setName("File " + fileName + ", " + queue.getName())
  .setTaskId(taskId)
  .setSlaveId(slaveID)
  .addResources(
     Resource.newBuilder().setName("cpus").setType(Value.Type.SCALAR)
       .setScalar(Value.Scalar.newBuilder().setValue(queue.getCpu())))
  .addResources(
     Resource.newBuilder().setName("mem").setType(Value.Type.SCALAR)
       .setScalar(Value.Scalar.newBuilder().setValue(queue.getMem())))
   .setData(ByteString.copyFrom(data.getBytes()))
   .setExecutor(ExecutorInfo.newBuilder(stdExecutor)).build();
```

Obrázek 6.3: Ukázka vytvoření úlohy pro exekutora

**Initialization** slouží pro načtení konfiguračního souboru a vytvoření úloh ze seznamu vstupních souborů. Vstupním parametrem aplikace je cesta ke konfiguračnímu souboru, který obsahuje informace o jednotlivých krocích, které budou soubory zpracovávat. Konfigurační soubor obsahuje tyto parametry:

`MODULEPATH` udává cestu ke složce modulů, které se budou spouštět pro jednotlivé kroky. Moduly se starají o spuštění přiřazeného souboru a zajišťují všechny potřebné věci, které jsou nutné pro spuštění programů ke zpracování. Dalším parametrem je `DBPATH`, který udává cestu k databázovému souboru, v němž se budou ukládat záznamy o zpracovaných souborech a jejich statistiky.

Dále jsou v bloku uvedeny jednotlivé kroky, přes které budou soubory procházet. Každý krok je identifikován pomlčkou, číslem a názvem. Krok musí obsahovat informace o potřebných zdrojích (CPU, paměť, doba zpracování). Potřebné zdroje CPU a paměť je možné zadat na dvě desetinná místa. Parametr `MODULE` obsahuje příkaz a název modulu, který se bude spouštět, tím je zajištěno spuštění libovolného programovacího jazyka. Posledním povinným parametrem je `INPUT` pro vstupní soubor, který obsahuje názvy souborů, jež se budou zpracovávat. Tento parametr je možné zaměnit za `DEPENDENCY` pro vytvoření závislosti na předchozím zpracováním. Závislost na konkrétním kroku je určena číslem, které musí odpovídat číslu v úvodní části bloku. Tímto způsobem je možné vytvořit libovolné řetězové zpracování, několik souběžných řetězových zpracování nebo jednotlivé nezávislé kroky.

Mezi volitelné parametry patří MPI pro spuštění MPI zpracování. Parametr `CONCURRENT` slouží pro postupné zpracování po jednom souboru. Posledním parametrem je `PART`, který slouží pro rozdělení zpracovávaných souborů do čtyř skupin a jejich oddělené zpracování. Tento parametr slouží pro aplikace, které zpracovávají více souborů současně. Pokud by se krok zpracovával jako celek, muselo by se čekat na dokončení všech potřebných souborů. Pokud takový krok rozdělíme, získáme lepší možnost při plánování úloh. Volitelné parametry je možné libovolně kombinovat a nastavit zpracování podle potřeby.

Konfigurační soubor je postupně zpracován a jsou vytvořeny fronty třídy `JobQueue`, které jsou naplněny získanými informacemi o náročnosti úloh a následujících frontách. Po vytvoření front se kontroluje zadání všech potřebných parametrů a závislostí mezi frontami.

```
MODULEPATH /tmp/module
DBPATH /tmp/sqlite.sql

# název fronty
-1 VERT

# povinné parametry
CPU 2.5
MEM 1500.0
TIME 00:15:00
MODULE python vert.py
INPUT /tmp/input

# volitelné parametry
CONCURRENT
MPI
PART
```

Obrázek 6.4: Ukázka konfiguračního souboru pro jeden krok

Po vytvoření front dojde k jejich naplnění úlohami. Pro všechny fronty, které mají v konfiguračním souboru parametr `INPUT`, je příslušný soubor převeden na úlohy a vložen do příslušné fronty.

**ContinueProcessing** je metoda, který je volána v případě spuštění aplikace s parametrem `-c`. Ten způsobí simulaci běhu po načtení souborů ke zpracování. Postupně se prochází frontami a jednotlivé soubory jsou vyhledány v databázi. Když je nalezen soubor, který byl úspěšně dokončen, je posunut do následující fronty metodou `finishTask`. Tímto způsobem obnovíme rozpracovanou práci. Pokud jsou některé soubory v databázi označeny jako chybové, dojde k jejich ponechání ve frontě. Tímto způsobem zajistíme pokračování ve zpracování po opravě programu bez nutnosti zasahovat do databáze.

## 6.7 Třída SimulatedAnnealing

Třída `SimulatedAnnealing` přijme parametry o zdrojích a seznam úloh, které je možné spustit. Prvním krokem je inicializace seznamu úloh, do kterého náhodně vybereme úlohy tak, aby nepřesáhly dostupné zdroje. Následuje prohledávání okolních stavů pro nalezení lepšího výsledku. Metoda `simulate` prohledává možnosti různých kombinací úloh a snaží se najít lepší výsledek. V každém kroku jsou nalezeny všechny vyhovující stavy. Každý stav se liší ve změně jednoho prvku. Z těchto stavů je jeden náhodně vybrán a porovnán, jestli přinese lepší výsledek. V případě lepšího výsledku je seznam úloh uložen a pokračujeme dalším cyklem algoritmu. V opačném případě, kdy vybraný prvek nepřinesl lepší výsledek, je pokračováno v další iteraci a v 1 % případů je vybrán jiný náhodný vzorek z nalezených kombinací, i pokud se jedná o horší uspořádání, se kterým se bude pokračovat. Tím je možné změnit cestu prohledávání s možností nalezení jiné kombinace úloh a nedojde k uváznutí v jedné možné kombinaci úloh. Po dokončení všech iterací je výsledkem seznam úloh s nej-

lepším ohodnocením, jež bylo nalezeno. Seznam úloh je nakonec předán třídě `MyScheduler`, která provede spuštění jednotlivých úloh a potvrzení zdrojů.

## 6.8 Vyhodnocení naměřených hodnot

Pro vyhodnocení statistik slouží skript napsaný v jazyku Python s názvem `stats.py`. Na vstupu potřebuje databázový soubor vytvořený aplikací. Všechny položky jsou načteny do slovníku, ze kterého je vytvořena webová stránka. Po načtení dat je po uživateli vyžádána cesta ke vstupním souborům pro každou nalezenou frontu v databázi. V zadaných cestách jsou nalezeny zpracované soubory a získána jejich velikost. Pro každou frontu je vytvořen přehled souborů s informacemi o spotřebované paměti, náročnosti na CPU, času zpracování, času stráveném na CPU a velikosti.

Z načtených dat jsou dále vytvořeny grafy, které zobrazují spotřebu paměti, dobu zpracování, závislost doby zpracování na velikosti souboru a paměťovou náročnost na velikosti souboru.

PARSE | TAG | SEC | VERT | DEDUP | Celkové statistiky

- Přehled
- Grafy
- Statistiky

**PARSING**

| Soubor | Paměť [MB] | CPU | Čas | Čas na CPU | Stav | Velikost |
|---|---|---|---|---|---|---|
| 1469257824226.79_20160723071024-00245 | 10547.0 | 1.07 | 00:06:52 | 00:02:23 | Success | 342474632 |
| 1469257824226.79_20160723071024-00269 | 10642.0 | 1.32 | 00:07:36 | 00:02:40 | Success | 378830104 |
| 1469257824226.79_20160723071024-00258 | 10528.0 | 1.51 | 00:07:16 | 00:02:37 | Success | 383292156 |
| 1469257824226.79_20160723071024-00252 | 10595.0 | 1.63 | 00:06:36 | 00:02:48 | Success | 404170887 |
| 1469257824226.79_20160723071024-00227 | 10538.0 | 1.38 | 00:07:20 | 00:02:52 | Success | 409548573 |
| 1469257824226.79_20160723071024-00236 | 10547.0 | 1.26 | 00:05:50 | 00:02:55 | Success | 415387257 |
| 1469257824226.79_20160723071024-00259 | 10539.0 | 1.01 | 00:06:33 | 00:02:40 | Success | 372827891 |
| 1469257824226.79_20160723071024-00240 | 10547.0 | 1.2 | 00:07:00 | 00:02:49 | Success | 396206283 |
| 1469257824226.79_20160723071024-00231 | 10622.0 | 0.83 | 00:07:41 | 00:03:00 | Success | 448161876 |
| 1469257824226.79_20160723071024-00234 | 10554.0 | 1.1188 | 00:07:10 | 00:02:57 | Success | 415004570 |
| 1469257824226.79_20160723071024-00260 | 10541.0 | 1.4 | 00:07:07 | 00:02:55 | Success | 409732837 |
| 1469257824226.79_20160723071024-00253 | 10654.0 | 1.52 | 00:06:09 | 00:03:09 | Success | 432868997 |
| 1469257824226.79_20160723071024-00232 | 10528.0 | 1.198 | 00:06:56 | 00:02:49 | Success | 401893577 |

Obrázek 6.5: Ukázka přehledu zpracovaných souborů

# Kapitola 7

# Testování a vyhodnocení systému

Tato kapitola se věnuje testování a celkovému vyhodnocení zpracování textových dat, které se provádí v několika krocích, a ověření, zda dojde ke zrychlení zpracování. V úvodu kapitoly budou představeny použité programy pro zpracování textu. Ve druhé části kapitoly představím výsledky měření zpracování. V závěru zhodnotím dosažené výsledky a statistiky získané během zpracování souborů.

### Vstupní data

Zpracování probíhalo na datech CommonCrawl s označením CC-2016-30. Jedná se cca o 51 TB dat webových stránek komprimovaných do desítek tisíc souborů. Takové množství je pro testování obrovské, a proto jsem zvolil malou část z této sady. Pro cluster Salomon jsem zvolil 230 souborů, což představuje cca 220 GB. Druhým testovacím prostředím je počítač Knot13, pro který jsem zvolil pouze 100 souborů o velikosti 98 GB, a to z důvodu slabšího hardwaru a pomalejšího zpracování tak, aby testování skončilo v rozumném čase.

### Kroky zpracování

- Vertikalizace – Do programu vstupují data ve formě Web Archive a z nich se vytvoří vertikální soubor, který má na každém řádku pouze jedno slovo, číslo, oddělovač nebo značku. Všechny ostatní části stránek jako menu, obrázky, tabulky, reklamy, formuláře a další prvky stránek jsou odstraněny. Dále se odstraňují HTML značky, dokud nezůstane jen důležitý text [29].

- Deduplikace – Při zpracovávání velkého množství dat, jako je CommonCrawl, dochází k situaci, kdy se do archivů dostanou duplicitní informace. Z tohoto důvodu je nutné provést odstranění duplicit. Agenti postupně procházejí informace a v podobě hashe je zasílají na server, který kontroluje, jestli se už daná informace objevila.

- POS – Jedná se o proces, který sekvenčně prochází slova a snaží se určit jejich slovní druhy na základě kontextu.

- PARSING – Zkoumá stavbu věty.

- SEC – Provádí obohacení textu o informace o pojmenovaných entitách, jako jsou osoby, místa, události apod.

## 7.1 Zpracování

Pro každé prostředí jsem zvolil tři sady testů. První sada bude zpracována standardním způsobem pomocí starého skriptu, kterým se doposud provádělo zpracování. Jedná se o zpracování postupné, kde se vždy provede stejný krok pro všechny soubory. Když jsou všechny dokončeny, přejde se k následujícímu kroku, dokud nejsou provedeny všechny kroky.

Zbylé dvě sady budou zpracovány novou aplikací. Poprvé dojde ke spuštění pouze s odhadem procesorové a paměťové náročnosti jednotlivých programů. Tím získáme statistiky, ze kterých se určí doporučené nastavení, a to se poté s drobnými úpravami použije pro poslední zpracování.

### 7.1.1 Zpracování na clusteru Salomon

Pro první spuštění byly nastaveny následující parametry:

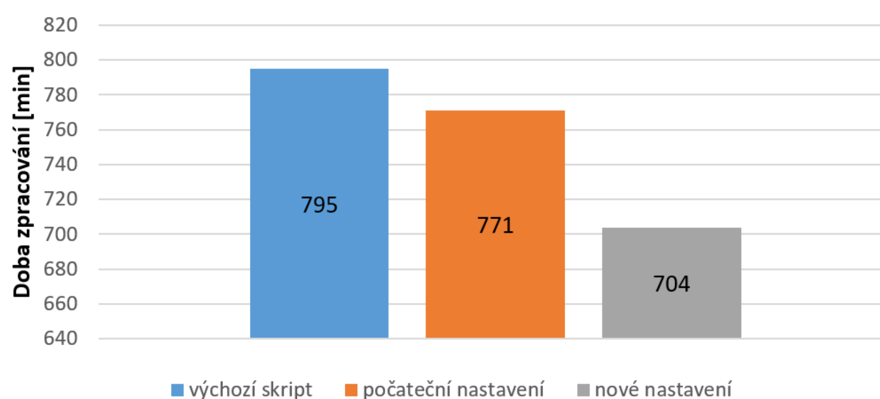
| | Vert | Dedup | Tag | Par | SEC |
|---|---|---|---|---|---|
| CPU | 2.0 | 4.0 | 1.0 | 1.0 | 1.0 |
| MEM | 1 000 | 1 000 | 8 000 | 13 000 | 30 000 |

Tabulka 7.1: Prvotní nastavení zpracování

Po dokončení prvního zpracování byl použit skript na vyhodnocení získaných informací a navrženy nové konfigurační hodnoty pro třetí test.

| | Vert | Dedup | Tag | Par | SEC |
|---|---|---|---|---|---|
| CPU | 1.15 | 4.0 | 0.71 | 0.71 | 1.15 |
| MEM | 600 | 1 000 | 6 000 | 12 000 | 29 000 |

Tabulka 7.2: Nové nastavení




Obrázek 7.1: Doba zpracování pro cluster Salomon

### 7.1.2 Zpracování na serveru Knot

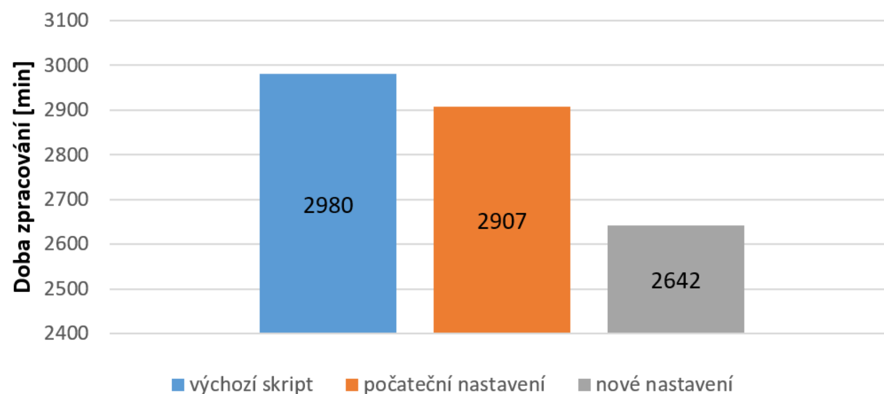
Pro server Knot13 byl použit stejný postup pro testování. Prvním měřením jsme získali konfigurační hodnoty pro poslední test. Počáteční nastavení:

| | Vert | Dedup | Tag | Par | SEC |
|---|---|---|---|---|---|
| CPU | 1.0 | 1.0 | 2.0 | 2.0 | 1.0 |
| MEM | 700 | 1 000 | 5 000 | 7 000 | 20 000 |

Tabulka 7.3: Prvotní nastavení pro server Knot13

| | Vert | Dedup | Tag | Par | SEC |
|---|---|---|---|---|---|
| CPU | 0.94 | 1.0 | 0.74 | 0.75 | 0.76 |
| MEM | 600 | 1 000 | 4 200 | 6 000 | 19 000 |

Tabulka 7.4: Nové nastavení pro zpracování na serveru Knot13




Obrázek 7.2: Doba zpracování pro server Knot13

### 7.1.3 Výsledky zpracování

V Grafu 7.1 jsou zobrazeny výsledky pro zpracování clusteru Salomon. Klasické zpracování pomocí skriptu skončilo za 13:15 hodin. Druhý test s počátečním nastaveném dokončil práci za 12:51 hodin, jeho nastavení je podobné skriptu z prvního testu. Poslední zpracování s upraveným nastavením dokázalo nejlépe využít dostupný hardware a dokončilo práci v čase 11:44. V tomto případě je zpracování zrychleno o 11,45 %, které přináší nová aplikace oproti původnímu zpracování.

Naměřené výsledky pro server Knot13 dopadly velmi podobně jako pro Salomon. Výsledky jsou zobrazeny na Grafu 7.2. Zpracování pomocí skriptu trvalo 49:40 hodin, test s počátečním nastavením trval 48:27 hodin a poslední test dokončil práci v čase 44:02. Zde je oproti současnému zpracování zrychlení o 11,35 %.

Zpracování novou aplikací je závislé na vhodném nastavení potřebných zdrojů pro jednotlivé kroky zpracování. S prvním nastavením se zpracování blíží k referenčnímu a zrychlení bylo pouze okolo 3 %. Po upřesnění nastavení se druhá sada testů zpracovala v průměru o 11,4 % rychleji než referenční zpracování.

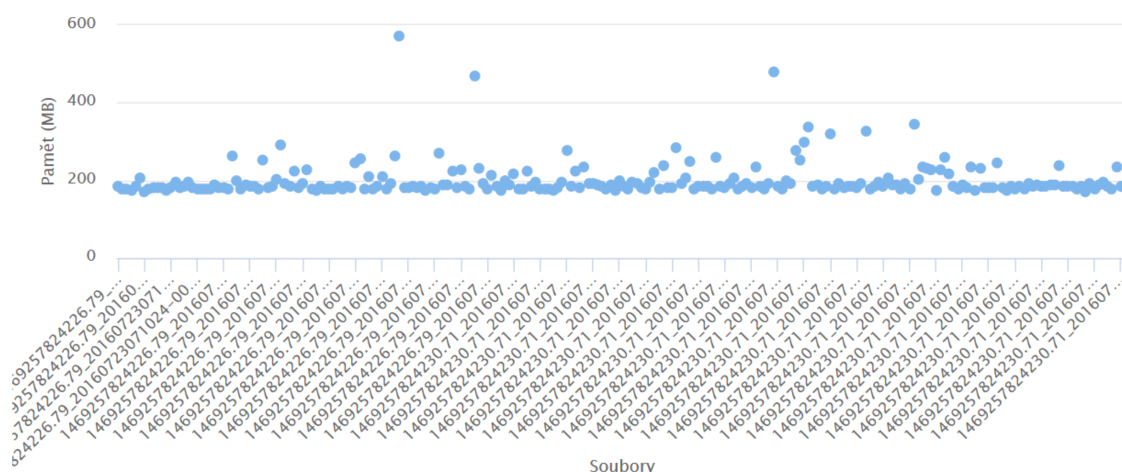
Aplikací se nám podařilo odstranit problém, kdy bylo nutné čekat mezi jednotlivými kroky na dokončení ostatních souborů. Toto zrychlení je proměnlivé a záleží na počtu prováděných kroků a také na počtu zpracovávaných souborů. Pokud by množina zpracovávaných souborů skončila přesně ve stejnou dobu, zrychlení by bylo minimální. Taková situace v praxi však nastane s malou pravděpodobností, protože soubory obsahují data s rozdílnou složitostí a soubory nebudou nikdy přesně stejné.

Druhým odstraněným problémem je náročnost některých kroků v jenom typu zdroje. Například pokud máme velkou paměťovou náročnost a zároveň nízkou procesorovou náročnost. Pokud takový krok pustíme pro všechny soubory současně, dojde k zatížení jen jednoho typu zdroje. Například krok SEC, který je schopný spotřebovat 20-30 GB paměti, je vhodným příkladem. Proto současné spuštění různě náročných kroků umožní spustit větší zátěž a lépe využít dostupné zdroje.

### 7.1.4 Naměřené statistiky

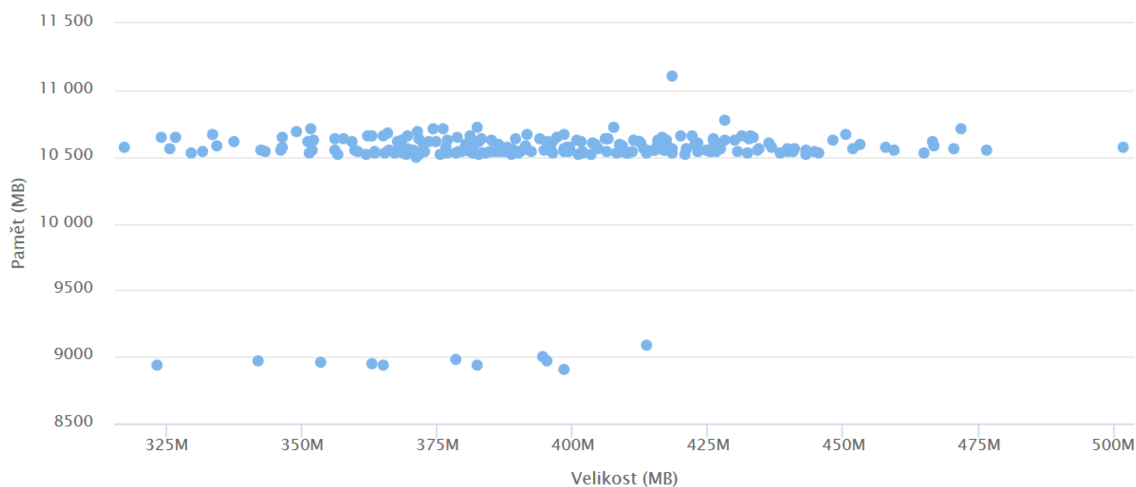
Ze získaných statistik jsou také vytvořeny grafy pro vylepšení spouštěných programů, nebo pro hledání souborů s rozdílnými vlastnostmi.

Z grafů se dají určit soubory, které mají nestandardní průběh zpracování. Například větší paměťovou náročnost, delší dobu zpracování a lineární závislosti na vstupu při zpracování. Nalezení těchto souborů je důležité a jejich analyzování může přinést důvod pro odlišné chování. Vytvoření optimalizací pro tyto soubory by mohlo vést k urychlení dalšího zpracování. Možností, jak využít statistiky, je zaměřit se na konkrétní program, jenž má největší nároky, a provést optimalizace. Můžeme vyměnit některý z programů, poté znovu provést zpracování a porovnat dobu zpracování nebo porovnat jejich nároky.

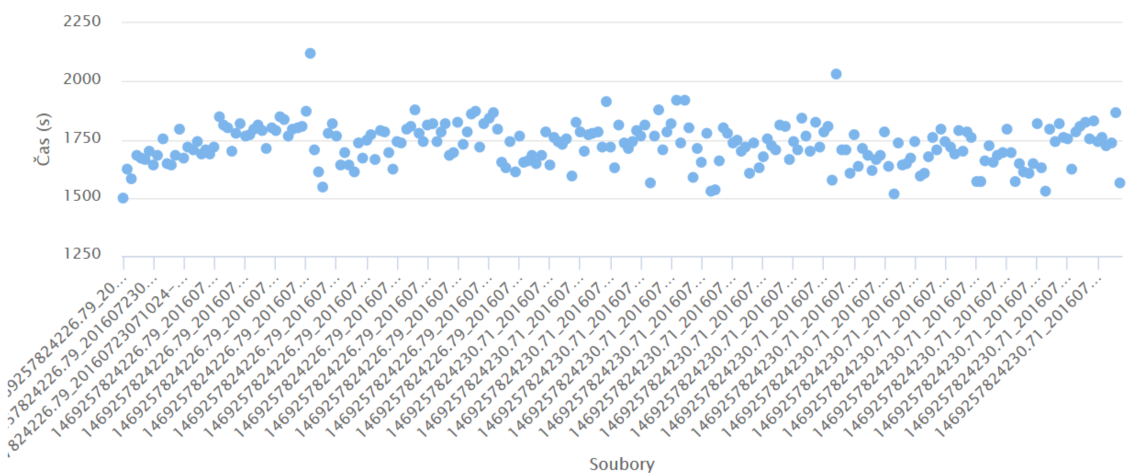



Obrázek 7.3: Spotřeba paměti pro jednotlivé soubory

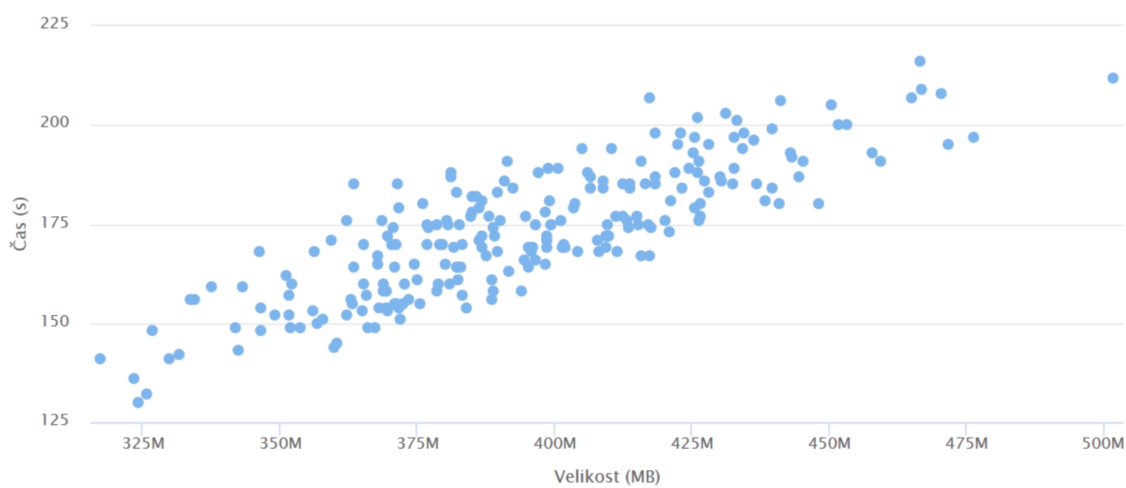
Když se podíváme na Graf 7.3 pro spotřebu paměti, vidíme, že tři soubory mají skoro trojnásobnou náročnost na paměť. Pokud analyzujeme tyto soubory a zjistíme, jaký je důvod nárůstu paměti, můžeme vytvořit potřebné optimalizace. Nastavení kroku zpracování se vztahuje na všechny jeho soubory a vyřešení takového problému, které přinese jen několik MB paměti, může v celkovém měřítku znamenat mnohem větší hodnotu.




Obrázek 7.4: Paměťová závislost na velikosti souboru




Obrázek 7.5: Doba zpracování pro jednotlivé soubory




Obrázek 7.6: Časová závislost na velikosti souboru

# Kapitola 8

# Závěr

Účelem práce bylo najít způsob efektivního využití zdrojů při zpracovávání rozsáhlých textových dat. V první části práce byl podrobně popsán způsob plánování úloh v distribuovaných systémech a představeny příklady těchto systémů. Dále je popsána organizace a struktura systému Apache Mesos, který je základem pro vytvořenou aplikaci.

Výsledkem praktické části práce je aplikace pro plánování úloh pro zpracování textu s maximálním využitím dostupného hardwaru. Jedním z požadavků na aplikaci byla i její univerzálnost, která je zajištěna implementací v Javě a konfiguračním souborem. Uživatel má možnost určit, jaké programy budou zařazeny do zpracování a může zvolit posloupnost jednotlivých kroků tak, aby získal požadovaná data.

Hlavním úkolem bylo zkrátit čas zpracování. To je dosaženo plánováním úloh v různých kombinacích tak, aby byl využit dostupný hardware na maximum. Tímto způsobem se podařilo zkombinovat spuštění náročných úloh s méně náročnými úlohami a celkové zpracování se zkrátilo téměř o 12 %. Zrychlení není konstantní, ale je závislé na počtu kroků a na rozdílnosti nároků pro jednotlivé programy.

Přes velkou snahu má aplikace určité nedostatky. Prvním je způsob monitorování spuštěných úloh. V případě, že modul využívá programy, které nejsou spuštěny z daného modulu, nejsou zaznamenány do měření. Příkladem je zpracování SEC použité při testování. Je založeno na agentovi, který už je spuštěn, a pouze je mu předán soubor na zpracování. Pokud taková situace nastane, není způsob pro detekování tohoto procesu a zahrnutí jeho statistik. Bylo by nutné udělat samostatné monitorování této fronty přímým zásahem do aplikace.

Druhým nedostatkem je zpracování na gridu, kde má každá stanice vlastní datový prostor. Zde je otázka, zda by se vyplatilo při pomalém zpracování některé stanice začít přesouvat soubory na rychlejší, tím odstranit čekání na pomalejší stanice. Tento způsob zpracování není na Salomonu nutný a jeho implementace by nejspíše vyžadovala samostatnou aplikaci. Muselo by se také rozhodnout, za kterých podmínek by docházelo k přesouvání souborů.

Vhodným rozšířením je vytvořit detailnější webové rozhraní napojené přímo na Apache Mesos. Zobrazovalo by aktuální využití uzlů a přehled aktuálně spuštěných úloh. Zajímavou možností by bylo vkládání dalších úloh do front při běhu aplikace.

# Literatura

[1] Apache Hadoop 3.0.0-alpha2 - 2013 Apache Hadoop YARN. [Online], 20.1.2017 [cit. 2017-05-10].
URL https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[2] Interactive JavaScript charts for your webpage | Highcharts. [Online], 19.12.2016 [cit. 2017-05-10].
URL https://www.highcharts.com/

[3] SYSSTAT. [Online], 15.8.2016 [cit. 2016-11-20].
URL http://sebastien.godard.pagesperso-orange.fr/index.html

[4] Beaulieu, A.: *Learning SQL*. O'Reilly Media, Inc., 2005, ISBN 0596007272.

[5] Coulouris, G.; Dollimore, J.; Kindberg, T.; aj.: *Distributed Systems: Concepts and Design*. USA: Addison-Wesley Publishing Company, páté vydání, 2011, ISBN 0132143011, 9780132143011.

[6] Dubhashi, D.; Das, A.: *Mastering Mesos*. Packt Publishing, 2016, ISBN 9781785885372.

[7] Fibich, P.; Matyska, L.; Rudová, H.; aj.: Model of grid scheduling problem. *Exploring Planning and Scheduling for Web Services, Grid and Autonomic Computing*, 2005: s. 17–24.

[8] Greenberg, D.: *Building Applications on Mesos: Leveraging Resilient, Scalable, and Distributed Systems*. O'Reilly Media, 2015, ISBN 9781491926581.
URL https://books.google.cz/books?id=WbElCwAAQBAJ

[9] Gupta, K.; Singh, M.: Heuristic based task scheduling in grid. In *International Journal of Engineering and Technology*, ročník 4, 2012, s. 254–260.

[10] Habernal, I.; Zayed, O.; Gurevych, I.: C4Corpus: Multilingual Web-size corpus with free license. In *Proceedings of LREC*, 2016, s. 914–922.

[11] Henick, B.: *HTML & CSS: The Good Parts: Better Ways to Build Websites That Work*. Animal Guide, O'Reilly Media, 2010, ISBN 9781449388751.
URL https://books.google.cz/books?id=gkBLI3xHYMUC

[12] Hindman, B.; Konwinski, A.; Zaharia, M.; aj.: Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA:

USENIX Association, 2011, s. 295–308.
URL http://dl.acm.org/citation.cfm?id=1972457.1972488

[13] Jackson, D. B.; Snell, Q.; Clement, M. J.: Core Algorithms of the Maui Scheduler. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP '01, London, UK, UK: Springer-Verlag, 2001, ISBN 3-540-42817-8, s. 87–102.
URL http://dl.acm.org/citation.cfm?id=646382.689682

[14] Jette, M. A.; Yoo, A. B.; Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, Springer-Verlag, 2002, s. 44–60.

[15] Joseph A, K.: A Comparison of Queueing, Cluster and Distributed Computing Systems. 1994.

[16] Kakadia, D.: *Apache Mesos Essentials*. Packt Publishing, 2015, ISBN 1783288760, 9781783288762.

[17] Keerthika, P.; Kasthuri, N.: A hybrid scheduling algorithm with load balancing for computational grid. *International Journal of Advanced Science and Technology*, ročník 58, 2013: s. 13–28.

[18] Kocjan, W.: Dynamic scheduling. State of the art report. Technická Zpráva T2002:28, December 2002.

[19] Li, M.; Baker, M.: *The Grid Core Technologies*. John Wiley & Sons, 2005, ISBN 0-47009-417-6.

[20] Lutz, M.: *Learning Python*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., druhé vydání, 2003, ISBN 0596002815.

[21] Manning, C. D.; Surdeanu, M.; Bauer, J.; aj.: The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, 2014, s. 55–60.

[22] Niemeyer, P.; Knudsen, J.: *Learning Java*. O'Reilly Media, Inc., 2005, ISBN 0596008732.

[23] Pacheco, P.: *An Introduction to Parallel Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., první vydání, 2011, ISBN 9780123742605.

[24] Pate, G.; Mehta, R.: A survey on various Task scheduling algorithm in cloud computing. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, ročník 3, č. 3, 2014: s. 715–717.

[25] Prajapati, H. B.; Shah, V. A.: Scheduling in grid computing environment. In *Advanced Computing & Communication Technologies (ACCT), 2014 Fourth International Conference on*, IEEE, 2014, s. 315–324.

[26] Schäfer, R.: CommonCOW: massively huge web corpora from CommonCrawl data and a method to distribute them freely under restrictive EU copyright laws. In *Proceedings of LREC*, 2016, s. 4500–4504.

[27] Schäfer, R.; DFG, L. W. C.: Processing and querying large web corpora with the COW14 architecture. In *Proceedings of the 3rd Workshop on Challenges in the Management of Large Corpora*, 2015, s. 28–34.

[28] Schwarzkopf, M.; Konwinski, A.; Abd-El-Malek, M.; aj.: Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, New York, NY, USA: ACM, 2013, ISBN 978-1-4503-1994-2, s. 351–364, doi:10.1145/2465351.2465386.
URL http://doi.acm.org/10.1145/2465351.2465386

[29] Švaňa, M.: *Čištění, extrakce textu a převod webových stránek do vertikálního formátu*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=18729

[30] Talbi, E.-G.; Muntean, T.: Hill-climbing, simulated annealing and genetic algorithms: a comparative study and application to the mapping problem. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, ročník 2, IEEE, 1993, s. 565–573.

[31] Thain, D.; Tannenbaum, T.; Livny, M.: Distributed Computing in Practice: The Condor Experience: Research Articles. *Concurr. Comput. : Pract. Exper.*, ročník 17, č. 2-4, Únor 2005: s. 323–356, ISSN 1532-0626, doi:10.1002/cpe.v17:2/4.
URL http://dx.doi.org/10.1002/cpe.v17:2/4

[32] Verma, A.; Pedrosa, L.; Korupolu, M.; aj.: Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA: ACM, 2015, ISBN 978-1-4503-3238-5, s. 18:1–18:17, doi:10.1145/2741948.2741964.
URL http://doi.acm.org/10.1145/2741948.2741964

[33] Wikipedia: Počítačový cluster. [Online], 15.3.2017 [cit. 20.11.2016].
URL https://cs.wikipedia.org/wiki/Počítačový_cluster

[34] Yousif, A.; Nor, S. M.; Abdualla, A. H.; aj.: Job Scheduling Algorithms on Grid Computing: State-of-the Art. *International Journal of Grid and Distributed Computing*, ročník 8, č. 6, 2015: s. 125–139.

[35] Zhu, Y.; Ni, L. M.: A survey on grid scheduling systems. *Department of Computer Science, Hong Kong University of science and Technology*, ročník 32, 2003.

# Příloha A

# Obsah přiloženého paměťového média

Na přiloženém CD se nachází:

- Skripty - Spouštěcí skripty pro Salomon a Knot prostředí, ukázka modulů pro zpracování textu, skript pro vyhodnocení statistik
- Aplikace-source - Zdrojové kódy aplikace
- Aplikace - Spustitelný jar soubor
- readme.txt - Popis spuštění jednotlivých programových částí
- zprava-source - Zdrojové kódy technické zprávy pro program LaTeX
- zprava.pdf - Technická zpráva ve formátu pdf

# Příloha B

# Plakát