**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE**
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-64329

# LÚŠTENIE HISTORICKÝCH ŠIFIER NA GRIDE

## DIPLOMOVÁ PRÁCA

**2017** **Bc. Martin Eliáš**

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE**
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-64329

# LÚŠTENIE HISTORICKÝCH ŠIFIER NA GRIDE

# DIPLOMOVÁ PRÁCA

| | |
|---|---|
| Študijný program: | Aplikovaná informatika |
| Číslo študijného odboru: | 2511 |
| Názov študijného odboru: | 9.2.9 Aplikovaná informatika |
| Školiace pracovisko: | Ústav informatiky a matematiky |
| Vedúci záverečnej práce: | Ing. Eugen Antal, PhD. |

**Bratislava 2017** **Bc. Martin Eliáš**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2016/2017
Evidenčné číslo: FEI-5384-64329

STU FEI

# ZADANIE DIPLOMOVEJ PRÁCE

| | |
|---|---|
| Študent: | **Bc. Martin Eliáš** |
| ID študenta: | 64329 |
| Študijný program: | aplikovaná informatika |
| Študijný odbor: | 9.2.9. aplikovaná informatika |
| Vedúci práce: | Ing. Eugen Antal, PhD. |
| Miesto vypracovania: | Ústav informatiky a matematiky |

Názov práce: **Lúštenie historických šifier na GRIDe**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Cieľom práce je vytvoriť nástroje na kryptoanalýzu klasických šifier v GRIDovom prostredí SIVVP. Vytvorené nástroje majú zahŕňať slovníkové útoky, útok hrubou silou a iné metódy kryptoanalýzy.

Úlohy:
1. Naštudujte problematiku kryptoanalýzy klasických šifier (brute-force, slovníkové útoky).
2. Naštudujte problematiku GRIDových výpočtov a dostupných technológií v rámci SIVVP.
3. Navrhnite a implementujte experimentálny výpočet na klastri SIVVP.
4. Otestujte a vyhodnoťte riešenie.

Zoznam odbornej literatúry:

1. Grošek, O. – Vojvoda, M. – Zajac, P. *Klasické šifry.* Bratislava : Vydavateľstvo STU, 2011. 214 s. ISBN 978-80-227-3486-8.
2. Sekaj, I. – Oravec, M. Paralelné evolučné algoritmy. In Kvasnička, V. – Pospíchal, J. – Návrat, P. – Lacko, P. – Trebatický, P. *Umelá inteligencia a kognitívna veda III.* Bratislava : Nakladateľstvo STU, 2011, s. 243–267. ISBN 978-80-227-3542-1.
3. Olson, E.: Robust Dictionary Attack of Short Simple Substitution Ciphers. Cryptologia 31-4, ISSN 0161-1194, 2007.

Riešenie zadania práce od: 19. 09. 2016

Dátum odovzdania práce: 19. 05. 2017

**Bc. Martin Eliáš**
študent



**prof. RNDr. Otokar Grošek, PhD.**
vedúci pracoviska

**prof. Dr. Ing. Miloš Oravec**
garant študijného programu

# SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

| | |
|---|---|
| Študijný program: | Aplikovaná informatika |
| Autor: | Bc. Martin Eliáš |
| Diplomová práca: | Lúštenie historických šifier na GRIDe |
| Vedúci záverečnej práce: | Ing. Eugen Antal, PhD. |
| Miesto a rok predloženia práce: | Bratislava 2017 |

Práca sa zameriava na vytvorenie nástrojov slúžiacich na kryptoanalýzu klasických šifier na gride a vývojového prostredia pre ďalších lúštiteľov. V prvých troch častiach sa práca venuje najmä teoretickým a čiastočne aj praktickým poznatkom z klasických šifier, MPI komunikácie a práce s gridom hpc.stuba.sk. Štvrtá kapitola práce sa zaoberá vývojovým prostredím a jeho použitím a tiež implementáciou paralelných genetických algoritmov. Posledné dve kapitoly majú experimentálne zameranie s cieľom lúštiť monoalfabetickú substitúciu pomocou paralelných genetických algoritmov na gride.

Kľúčové slová: historické šifry, grid, MPI, paralelné genetické algoritmy

# ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

| | |
|---|---|
| Study Programme: | Applied Informatics |
| Author: | Bc. Martin Eliáš |
| Master's thesis: | Solving of historical ciphers on GRID like systems |
| Supervisor: | Ing. Eugen Antal, PhD. |
| Place and year of submission: | Bratislava 2017 |

The main aim of the thesis is to develop tools that would serve for cryptanalyzing the classical ciphers on the grid and the development environment for future cryptanalysts. The first three chapters are dedicated to theoretical and partly practical information of classical ciphers, MPI communication and the work with the grid hpc.stuba.sk. The fourth chapter deals with the development environment and its use as well as the implementation of parallel genetic algorithms. The two final chapters are of an experimental nature and their main aim is to decipher the mono-alphabetic substitution with the help of the parallel genetic algorithms.

Keywords: historical ciphers, grid, MPI, parallel genetic algorithms

# Poďakovanie

Chcem sa poďakovať najmä Ing. Eugenovi Antalovi, PhD., za odbornú pomoc, rady a usmernenia pri písaní tejto práce. Vďaka patrí aj HPC centru Slovenskej Technickej Univerzity v Bratislave, ktorá je súčasťou slovenskej infraštruktúry High Performance Computing (projekt SIVVP, ITMS kód 26230120002, podporovaný Európskym fondom rozvoja regiónov (ERDF), za výpočtový čas a poskytnuté zdroje. V neposlednom rade patrí vďaka aj mojej rodine a snúbenici za trpezlivosť a podporu počas môjho štúdia.

# Obsah

**Úvod** **1**

**1 Klasické šifry** **2**

1.1 História . . . 2

1.2 Charakteristika klasických šifier . . . 3

1.3 Počítačové lúštenie klasických šifier . . . 4

1.3.1 Útok hrubou silou . . . 5

1.3.2 Slovníkový útok . . . 6

1.3.3 Horolezecký algoritmus . . . 6

1.3.4 Genetické algoritmy . . . 6

1.3.5 Paralelné genetické algoritmy . . . 9

**2 Grid** **10**

2.1 hpc.stuba.sk . . . 10

2.2 Príkazy . . . 11

2.2.1 module . . . 11

2.2.2 qstat . . . 12

2.2.3 qfree . . . 13

2.3 Výpočtové fronty . . . 14

2.3.1 Príklad sériovej úlohy . . . 14

2.3.2 Príklad paralelnej úlohy . . . 15

**3 MPI** **17**

3.1 Point-to-point komunikácia . . . 17

3.2 Blokujúca komunikácia . . . 18

3.3 Neblokujúca komunikácia . . . 20

3.4 Dynamická alokácia . . . 21

3.5 Serializácia dátových typov . . . 22

**4 Návrh a implementácia** **23**

4.1 Štruktúra prostredia . . . 23

4.2 Technológie . . . 24

4.3 Skript build.sh . . . 25

4.4 PGA modul . . . 26

4.4.1 Genetický algoritmus . . . 28

4.4.2 Serializácia chromozómov . . . . . 29
4.4.3 Mpi migrátor a topológie . . . . . 31
4.5 Príklad použitia . . . . . 32
4.5.1 Lokálny vývoj . . . . . 32
4.5.2 Spustenie na gride hpc.stuba.sk . . . . . 35

**5 Experiment s GA** **37**
5.1 Distribúcia parametrov . . . . . 37
5.2 Výsledky experimentu . . . . . 40

**6 Experiment s PGA** **45**
6.1 Výsledky experimentu . . . . . 46

**Záver** **49**

**Zoznam použitej literatúry** **50**

**Prílohy** **I**

**A Štruktúra elektronického nosiča** **II**

# Zoznam obrázkov a tabuliek


Obrázok 1 Štruktúra GA (vytvorené na základe [2]) . . . . . 8
Obrázok 2 Štruktúra PGA s migračným modelom (vytvorené na základe [1]) 26
Obrázok 3 Závislosť úspešnosti lúštenia od dĺžky ZT (10000 iterácií, 10 jedincov) . . . . . 41
Obrázok 4 Závislosť úspešnosti lúštenia od dĺžky ZT (10000 iterácií, 50 jedincov) . . . . . 41
Obrázok 5 Závislosť úspešnosti lúštenia od dĺžky ZT (50000 iterácií, 100 jedincov) . . . . . 42
Obrázok 6 Závislosť úspešnosti lúštenia od dĺžky ZT (schéma B) . . . . . 43
Obrázok 7 Závislosť úspešnosti lúštenia od dĺžky ZT (schéma C) . . . . . 43
Obrázok 8 Závislosť úspešnosti lúštenia od dĺžky ZT (schéma E) . . . . . 44
Obrázok 9 Závislosť úspešnosti lúštenia od dĺžky ZT (schéma J) . . . . . 44
Obrázok 10 Topológie paralelných genetických algoritmov [1] . . . . . 45
Obrázok 11 Závislosť úspešnosti lúštenia od dĺžky ZT (topológia b/3, 10 jedincov) . . . . . 46
Obrázok 12 Závislosť úspešnosti lúštenia od dĺžky ZT (topológia d/3, 10 jedincov) . . . . . 47
Obrázok 13 Závislosť úspešnosti lúštenia od dĺžky ZT (topológia b/3, 20 jedincov) . . . . . 47
Obrázok 14 Závislosť úspešnosti lúštenia od dĺžky ZT (topológia b/5, 50 jedincov) . . . . . 48
Obrázok 15 Závislosť úspešnosti lúštenia od dĺžky ZT (topológia b/11, 100 jedincov) . . . . . 48

Tabuľka 1 Dostupné umiestnenia pre používateľa na hpc.stuba.sk . . . . . 11
Tabuľka 2 Výpočtové fronty a ich obmedzenia . . . . . 14
Tabuľka 3 Dátové typy v MPI a ich C ekvivalenty . . . . . 20
Tabuľka 4 Operácie GA . . . . . 27
Tabuľka 5 Schémy GA . . . . . 38

# Zoznam skratiek

| | |
|---|---|
| **BASH** | Bourne Again SHell |
| **CPU** | Central Processing Unit |
| **GA** | Genetické Algoritmy |
| **GPFS** | General Parallel File System |
| **GPU** | Graphics Processing Unit |
| **HDD** | HardDisk Drive |
| **LAN** | Local Area Network |
| **MPI** | Message Passing Interface |
| **OpenMP** | Open Multi-Processing |
| **OpenMPI** | Open Message Passing Interface |
| **PBS** | Portable Batch System |
| **PGA** | Paralelné Genetické Algoritmy |
| **RAM** | Random Access Memory |
| **SSH** | Secure SHell |
| **VPN** | Virtual Private Network |

# Zoznam výpisov


1 module avail . . . . . 11
2 qstat . . . . . 12
3 qstat -u 3xelias . . . . . 12
4 qfree . . . . . 13
5 uloha1.pbs . . . . . 14
6 uloha2.pbs . . . . . 15
7 Point-to-point komunikácia . . . . . 17
8 MPI_Send . . . . . 19
9 MPI_Recv . . . . . 19
10 Neblokujúca komunikácia . . . . . 20
11 Štruktúra prostredia . . . . . 23
12 Genetický algoritmus . . . . . 28
13 Abstraktná trieda Migrator . . . . . 31
14 Kód MpiMigrator-a . . . . . 31
15 HelloGrid/src/main.cpp . . . . . 33
16 Pseudokód ditribúcie parametrov GA . . . . . 39

# Úvod

Výber diplomovej práce *Lúštenie historických šifier na GRIDe* bol podmienený osobným zájmom o túto problematiku. Hlavným cieľom tejto práce je vytvorenie nástrojov na kryptoanalýzu klasických šifier v gridovom prostredí SIVVP. V tejto práci implementujeme útok pomocou paralelných genetických algoritmov. Tematikou paralelných výpočtov sa zaoberala aj práca [1] a čiastočne aj práca [2], z ktorých sme čerpali niekoľko základných poznatkov o paralelných genetických algoritmoch. Ďalším cieľom tejto práce je vyvinúť vývojové prostredie slúžiace pre ďalších potenciálnych lúštiteľov, ktorí by mohli potrebovať ku svojej práci gridové prostredie.

Štruktúra diplomovej práce sa skladá zo šiestich kapitol. Prvá kapitola slúži ako teoretický úvod do problematiky klasických šifier počnúc históriou, charakteristikou a počítačovým lúštením klasických šifier. V tejto kapitole predstavíme aj niektoré základné útoky.

Druhá kapitola práce sa zameriava na predstavenie gridu *hpc.stuba.sk*, pričom naším cieľom bude predstaviť základy práce so spomínaným prostredím. Okrem toho si poukážeme na poskytovanú funkcionalitu tohto prostredia a predstavíme niekoľko základných typov úloh.

V tretej kapitole si uvedieme základné princípy MPI komunikácie, ktoré budeme využívať pri medziprocesorovej komunikácii na gride. Taktiež poskytneme aj niekoľko principiálnych ukážok komunikácie a spomenie si dva základné problémy súvisiace s výmenou dát pomocou MPI.

Nasledujúca kapitola pozostáva zo stručného návrhu a požiadaviek kladených na vývojové prostredie. Táto kapitola má praktické zameranie a predstavíme si v nej použité technológie, základ vývojového prostredia ktorým je skript build.sh. V nasledujúcom kroku predstavíme vývoj a implementáciu paralelných genetických algoritmov. Táto kapitola má slúžiť aj ako manuál pre ďalších používateľov tohto prostredia.

V piatej kapitole vykonáme experiment s genetickými algoritmami a ukážeme prístup zvolený pri paralelizácii programu.

Záverečná kapitola pozostáva z ďalšieho experimentu súvisiaceho s paralelnými genetickými algoritmami, pri ktorom vychádzame z kapitoly päť.

# 1 Klasické šifry

Táto kapitola sa zaoberá históriou a stručným prehľadom klasických šifier. Spomenieme si aj niektoré základné útoky na klasické šifry. Táto kapitola bola vypracovaná na základe literatúry [3], [2] a [1].

## 1.1 História

História klasických šifier a utajovania písomného textu je pravdepodobne tak stará ako samotné písmo. Písmo, v podobe akej ho poznáme a používame dnes, pravdepodobne pochádza asi spred 3000 rokov pred Kristom a za jeho objaviteľov sa považujú Feničania. V niektorých prípadoch predstavovalo už použitie písma utajenie samotného textu. Príkladom môžu byť egyptské hieroglyfy alebo klinové písmo používané v Mezopotámii. Iným príkladom môžu byť semitské jazyky, ktoré sú charakteristické používaním iba spoluhlások bez použitia samohlások, pretože tie zaviedli až Aramejci a po nich následné Gréci, aby pomocou nich boli schopní rozlíšiť jazyky. Aj diakritika ako taká, má schopnosť rozlišovať významy slov, čo si ale až do 15. storočia nikto nevšímal, až pokiaľ ju Arabi nezačali používať pri kryptoanalýze rôznych šifier.

Z historického hľadiska nie je možné presne zoradiť ako jednotlivé šifry vznikali, pretože súčasne vznikali na viacerých miestach sveta. Komunikácia a s ňou spojené šírenie informácií nebolo také rýchle ako dnes, až do roku 1440, keď Johan Guttenberg vynašiel kníhtlač, čo zjednodušilo výmenu a uchovávanie informácií.

Ku kryptografii ako aj k rôznym iným vedným disciplínam prispelo v minulosti staré Grécko. Jedným z najvýznamnejších príspevkov starých Grékov bolo rozšírenie abecedy a písomného prejavu. Gréci písmo prebrali od Feničanov, ktorí na rozdiel od Egypťanov používali jednoduchšie písmo.

V Európe vďaka rozšíreniu abecedy začali vznikať aj prvé šifry, medzi ktoré patrí napríklad *Cézarova šifra*, ktorá vznikla v Rímskej ríši. Iným príkladom môže byť transpozičná šifra *Skytalé*, ktorá bola používaná v Sparte.

Pád Rímskej ríše spôsobil úpadok kryptografie, ktorý trval až do obdobia stredoveku. Typickým znakom kryptografie v tomto období bolo napríklad písanie odzadu alebo vertikálne, používanie cudzích jazykov, alebo vynechávanie samohlások.

V stredoveku kvôli bojom medzi pápežmi Ríma a Avignonu, bola kryptografia zdokonalená a začali sa používať rôzne kódy a nomenklátory. Ich charakteristickým znakom bolo zamieňanie písmen alebo nahradzovanie mien a titulov osôb v správach. V tomto období zabezpečovanie utajenia správ pokročilo až na takú úroveň, že na doručovanie

správ boli použití špeciálne vycvičení kuriéri.

V prvej polovici 20. storočia ľudia, ktorí pracovali v oblasti utajovanej komunikácie verili, že na to, aby bola zabezpečená utajovaná komunikácia musí byť utajený kľúč a okrem neho aj šifrovací algoritmus. Toto ale odporovalo Kerckhoffovmu princípu, ktorý hovorí že: „Bezpečnosť šifrovacieho algoritmu musí závisieť výlučne na utajení kľúča a nie algoritmu“. Okrem toho sformuloval aj niekoľko požiadaviek na kryptografický systém, medzi ktoré patria:

1. Systém musí byť teoreticky, alebo aspoň prakticky bezpečný.
2. Narušenie systému nesmie priniesť ťažkosti odosielateľovi a adresátovi.
3. Kľúč musí byť ľahko zapamätateľný a ľahko vymeniteľný.
4. Zašifrovaná správa musí byť prenášateľná telegrafom.
5. Šifrovacia pomôcka musí byť ľahko prenosná a ovládateľná jedinou osobou.
6. Systém musí byť jednoduchý, bez dlhého zoznamu pravidiel, nevyžadujúci nadmerné sústredenie.

Tieto princípy sú popísané v pôvodnej publikácii od Kerckhoffa [4].

Existovala ale aj iná skupina vedcov, medzi ktorých patril aj Lester S. Hill, ktorý si uvedomoval, že kryptológia je úzko spätá s matematikou. V roku 1941 si na Hillových prácach zakladal A. Adrian Albert, ktorý pochopil, že v šifrovaní je možné použiť viacero algebraických štruktúr. Neskôr toto všetko usporiadal a zdokonalil Claude E. Shannon, čo možno považovať za ukončenie éry klasických šifier.

## 1.2 Charakteristika klasických šifier

Na rozdiel od moderných šifier, ktoré sa používajú dnes, sú tie klasické rozdielne v niektorých hlavných črtách. Môžeme spomenúť niekoľko rozdielov:

- Šifrovanie a dešifrovanie klasickej šifry možno realizovať zväčša pomocou papiera a ceruzky alebo nejakej mechanickej pomôcky.
- V dnešnej dobe aj vďaka rozšírenému použitiu počítačov stratila väčšina týchto algoritmov svoj význam.
- Utajuje sa algoritmus a aj kľúč a neuplatňuje sa Kerckhoffov princíp.
- Na rozdiel od moderných šifier sa používajú malé abecedy.

- V klasických šifrách je otvorený text, zašifrovaný text a kľúč v abecede reálneho jazyka, pričom v moderných šifrách sa používa binárne kódovanie.
- Na klasické šifry sa zväčša dá použiť štatistická analýza textu.

Zo spomenutých charakteristík existujú aj výnimky. Napríklad pri Vigenerovej šifre sa algoritmus neutajoval. To platí aj pre Vernamovu šifru, ktorá okrem toho používa navyše binárne znaky. Vernamova šifra je perfektne bezpečná v podľa Shannonovej teórie.

Klasické šifry môžeme rozdeliť do niekoľkých základných kategórií:

- **Substitučné šifry.** V prípade, že šifra permutuje znaky zdrojovej abecedy, hovoríme o monoalfabetickej šifre. Ako príklad môžeme uviesť šifru Atbaš prípadne Cézarovu šifru, alebo iné. V inom prípade, ak sa aplikuje viacero permutácií podľa polohy znaku v otvorenom texte, tak hovoríme o polyalfabetickej šifre. Príkladom je Vigenerova šifra. Ďalším prípadom je polygramová šifra, kde sa z otvoreného textu najprv vytvoria bloky, na ktoré sa potom aplikuje nejaká permutácia.
- **Transpozičné šifry.** Transpozičné šifry sú vlastne blokové šifry, ktoré pri šifrovaní a dešifrovaní aplikujú pevne zvolenú permutáciu na každý blok otvoreného, zašifrovaného textu.
- **Homofónne šifry.** Homofónne šifry sú šifry, ktoré majú znáhodnený zašifrovaný text. Tieto šifry sa snažia zabrániť frekvenčnej analýze textu.
- **Substitučno-permutačné šifry.** Ak aplikujeme viacero substitučných a permutačných šifier na otvorený text, tak hovoríme o substitučno-permutačných šifrách. Šifrovanie prebieha tak, že sa blok otvoreného textu rozdelí na menšie bloky, na ktoré je potom aplikovaná substitúcia a permutácia. Substitúcia zabezpečuje konfúziu a permutácia difúziu.

## 1.3 Počítačové lúštenie klasických šifier

Počítače a lúštenie šifier sú v dnešnej dobe neoddeliteľnou súčasťou. Výkon počítačov vplyvom Mooreovho zákona [1] neustále narastá, čo má za následok vznik nových metód, ktoré sa snažia využiť dostupný výkon.

„Výpočtový výkon súčasných počítačov môže byť použitý na vykonanie čiastočnej, alebo až úplnej kryptoanlýzy klasických šifier“. Takúto automatizovanú kryptoanalýzu môžeme rozdeliť do troch kategórií:

[1] Moorov zákon je empirické pravidlo, ktoré hovorí, že zložitosť integrovaných obvodov sa zdvojnásobuje každých 18 až 24 mesiacov, pričom cena ostáva konštantná.

1. Prvou kategóriou je počítačom asistovaná kryptoanalýza, pri ktorej lúštiteľ využíva počítač ako pomôcku na zjednodušenie niektorých úkonov.

2. Druhou kategóriou je poloautomatická krytoanalýza, pri ktorej počítač prehľadáva priestor riešení, avšak rozhodujúce kroky sú prenechané na kryptoanalytika.

3. Poslednou, treťou kategóriou je automatická kryptoanalýza, pri ktorej lúštiteľ zadá zašifrovanú správu počítaču, a ten mu následne poskytne jedno, prípadne aj viac najlepších riešení.

Automatickou kryptoanalýzou sa budeme zaoberať aj v tejto práci. Pri automatickej kryptoanalýze zohráva veľkú úlohu analýza a ohodnocovanie textov, pretože niektoré spôsoby lúštenia šifier, napríklad útok hrubou silou, produkujú veľké množstvo možných riešení, ktoré treba nejakým spôsobom ohodnotiť.

Na automatické ohodnocovanie textov sa používa ohodnocovacia funkcia, pomocou ktorej môžeme stanoviť určitý stupeň kvality, takzvané skóre. Podľa skóre potom môžeme niektoré texty zahodiť a iné posunúť na ďalšie posúdenie. Jednoduchú ohodnocovaciu funkciu by sme mohli skonštruovať napríklad pomocou slovníka. Existujú ale aj iné, lepšie metódy, napríklad n-gramy, ktoré využívajú frekvencie dvojíc, trojíc, n-tíc znakov jazyka.

Pri vhodne zvolenej ohodnocovacej funkcii začnú vznikať v priestore riešení takzvané extrémy. Na to, aby sme v priestore riešení našli globálny extrém, nepotrebujeme prehľadať všetky riešenia. Existujú algoritmy, ktoré dokážu nájsť globálny extrém rýchlejšie. Medzi takéto algoritmy patria napríklad: horolezecký algoritmus, simulované žíhanie, genetické algoritmy a iné.

### 1.3.1 Útok hrubou silou

Útok hrubou silou (bruteforce) je typ útoku, ktorý sa snaží zlomiť kľúč tak, že sa prehľadáva celý priestor kľúčov. Aby bol takýto útok možný a prakticky realizovateľný, priestor prehľadávaných kľúčov nesmie byť väčší ako hranica daná dostupnými prostriedkami alebo časom potrebným na riešenie.

Pre ilustráciu si uveďme jednoduchý príklad. Majme zašifrovaný text *VECDOXSORSCDYBSMUIMRCSPSOBXKQBSNO*, ktorý vieme, že bol zašifrovaný Cézarovou šifrou. Pre získanie otvoreného textu potrebujeme vyskúšať všetkých 26 možností posunov, čo je v tomto prípade kľúč tak, aby sme dostali zmysluplný text.

```
kluc 1
VECDOXSORSCDYBSMUIMRCSPSOBXKQBSNO
WFDEPYTPSTDEZCTNVJNSDTQTPCYLRCTOP
```

```
kluc 2
VECDOXSORSCDYBSMUIMRCSPSOBXKQBSNO
XGEFQZUQTUEFADUOWKOTEURUQDZMSDUPQ

... // dalsie kluce 4..26
```

Po prezretí všetkých možností by sme zistili, že kľúč 16 sa dešifruje na *LUSTENIE-HISTORICKYCHSIFIERNAGRIDE*.

### 1.3.2 Slovníkový útok

Slovníkový útok narozdiel od útoku hrubou silou skúša iba niektoré možnosti z vopred pripraveného slovníka kľúčov.

Ukážme si ako by v princípe mohol fungovať slovníkový útok na šifru Vigenere. Nech zašifrovaný text je *SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW*. Útočník má k dispozícii slovník slov *ABC, HESLO, ....*

```
kluc: ABC
SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW
JYXQJUZIJWMJXCJVFQAHQMKFJYEEIFCTN

kluc: HESLO
SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW
LUSTENIEHISTORICKYCHSIFIERNAGRIDE
```

V tomto príklade bol kľúč *HESLO* a text sa dešifroval na *LUSTENIEHISTORIC-KYCHSIFIERNAGRIDE*.

### 1.3.3 Horolezecký algoritmus

Horolezecký algoritmus patrí medzi základné optimalizačné algoritmy. Jeho základnou myšlienkou je, že na vrchol kopca sa môžeme dostať najkratšou cestou keď pôjdeme vždy najstrmším smerom nahor.

Algoritmus sa na začiatku vždy inicializuje náhodným kľúčom. Následne sa vygeneruje množina nových potencionálnych kandidátov na riešenie pomocou základných zmien. Ak je kľúč napríklad permutácia, zmenou môže byť výmena dvoch prvkov v kľúči. Jednotlivým kľúčom je potom priradené skóre pomocou ohodnocovacej funkcie. Za aktuálny kľúč sa bude považovať ten, ktorý má najlepšie skóre. Výsledky tohto algoritmu ovplyvňuje voľba ohodnocovacej funkcie, ktorá ľahko môže ostať v lokálnom extréme, z ktorého sa nevie dostať.

### 1.3.4 Genetické algoritmy

Genetické Algoritmy (GA) patria medzi najčastejšie používaných predstaviteľov evolučných výpočtových techník. Genetické algoritmy sa snažia napodobniť biologické pro-

cesy v prírode.

Základnými objektami sú gén, reťazec a populácia. Nad týmito objektami sa vykonávajú operácie. Medzi základné operácie patria výber, mutácia a kríženie.

**Gén** je základnou stavebnou jednotkou reťazca a predstavuje elementárne vlastnosti jedinca. Zvyčajne je reprezentovaný číselne, alebo nejakým symbolom z abecedy.

**Reťazec (chromozóm)** je postupnosť génov, respektíve znakov, ktoré predstavujú zvolené parametre alebo vlastnosti jedinca z problémovej oblasti. V tomto prípade reťazec predstavuje dešifrovací kľúč.

**Populácia** je skupina reťazcov zvoleného počtu. Veľkosť populácie sa počas riešenia genetického algoritmu môže meniť.

**Generácia** predstavuje populáciu GA v niektorej výpočtovej fáze, prípadne môže reprezentovať poradové číslo výpočtového cyklu.

**Účelová funkcia** vypočítava skóre každého jedinca v populácii a je mierou toho, čo chceme maximalizovať, prípadne minimalizovať.

**Fitness** je v evolučných výpočtoch pojem predstavujúci mieru úspešnosti jedincov. V prípade maximalizačnej úlohy je to najväčšia hodnota účelovej funkcie. Naopak, v prípade minimalizačnej úlohy je to najmenšia hodnota.

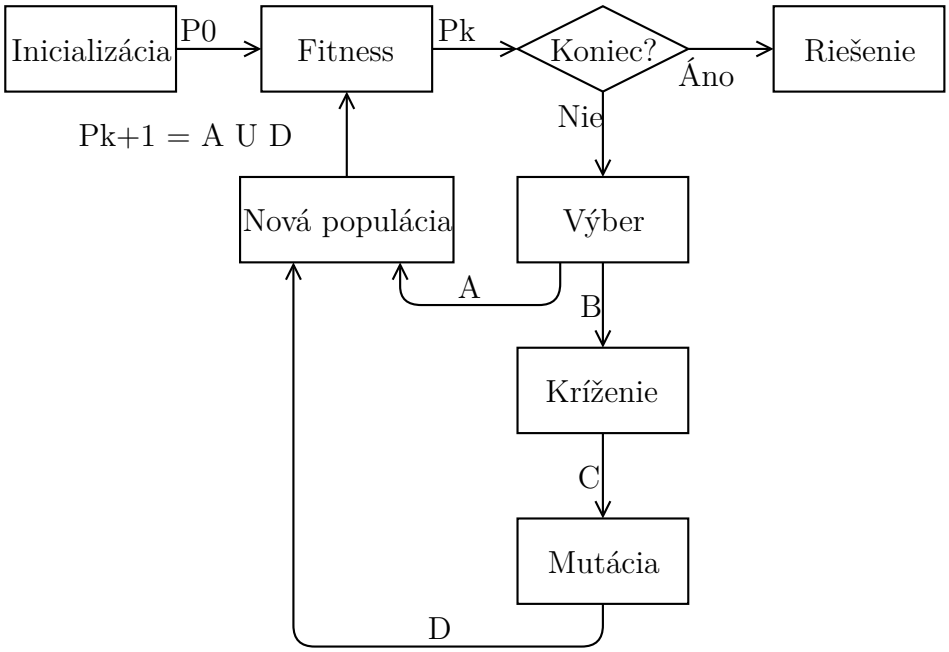
**Výber** je proces, ktorý vyberie niektorých jedincov z populácie na základe zvolenej stratégie. Vybraní jedinci potom vstupujú do operácií kríženia, mutácie alebo bez zmeny pokračujú do ďalšej generácie. Existuje viacero stratégií výberu jedincov, avšak základnou ideou je, aby lepší jedinci prežili a postupne vytlačili slabších jedincov z populácie.

**Mutácia** znamená náhodnú zmenu génu v reťazci, prípadne viac zmien v celej populácii. Gén zmení svoju hodnotu na inú, náhodne zvolenú hodnotu z prehľadávaného priestoru. Mutácia je základnou hybnou silou genetických algoritmov. Umožňuje nachádzať nové riešenia, ktoré sa v populácii ešte neobjavili.

**Kríženie** je operácia, pri ktorej sa dva náhodne zvolené rodičovské jedince rozdelia v nejakom bode a vymenia si svoje gény. Krížením vznikajú noví, odlišní potomkovia, ktorí nesú niektoré znaky oboch rodičov.

Skôr než sa GA začne realizovať, treba určiť spôsob zakódovania parameterov jednotlivých objektov. Pri klasických šifrách sa gény zvyčajne reprezentujú pomocou abecedných znakov alebo čísiel. Ďalším krokom je určiť prehľadávaný priestor a následne formulovať účelovú funkciu. Dôležité je takisto určiť veľkosť populácie, ktorá sa doporučuje v rozsahu

od 10 do 100 jedincov. Pri malých populáciách býva nedostatočný priestor pre rôznorodosť (diverzitu) genetických informácií, naopak pri veľkých populáciách sa už nedosahuje lepší efekt.




Obrázok 1: Štruktúra GA (vytvorené na základe [2])

Princíp genetických algoritmov možno vidieť aj na obrázku 1, ktorý je realizovaný nasledovne:

**Inicializácia** Vygeneruje počiatočnú populáciu $P_0$.

**Fitness** Vyhodnocuje populáciu pomocou účelovej funkcie pre aktuálnu populáciu $P_k$.

**Koniec?** V tejto fáze sa testujú ukončujúce podmienky genetického algoritmu. V prípade, že algoritmus skončil, vyberie sa najlepší jedinec z aktuálnej populácie, ktorý predstavuje výsledné riešenie GA.

**Výber** Ak algoritmus ešte neskončil, nasleduje výber dvoch skupín jedincov. Najprv sa vyberie jeden, prípadne viac najlepších jedincov, ktorí sa bez zmeny skopírujú do novej populácie (skupina $A$). Tieto reťazce zabezpečia monotónnu konvergenciu, čo znamená, že úloha v ďalšom kroku nebude mať horšie výsledky ako v tom predchádzajúcom. Ďalšie jedince vybraté iným spôsobom sú skopírované do operácie kríženia (skupina $B$).

**Kríženie** V operácii kríženia sa náhodne spárujú rodičovské reťazce so skupiny $B$, z ktorých vznikne rovnaký počet potomkovských reťazcov. Potomkovia vytvoria skupinu $C$, ktorá pokračuje do operácie mutácie.

**Mutácia** spôsobí, že náhodne zvoleným jedincom zmenia niektoré náhodne zvolené gény a vznikne populácia $D$.

**Nová populácia** $P_{k+1}$ vzniká zjednotením populácií $A \cup D$. Genetický algoritmus pokračuje bodom 2.

Predstavená schéma je len príklad ako by GA mohol vyzerať, v skutočnosti však môžeme uvažovať o viacerých nastaveniach, ktoré vplývajú na GA. Niektoré uvedieme v experimentálnej časti tejto práce.

### 1.3.5 Paralelné genetické algoritmy

Paralelné Genetické Algoritmy (PGA) možno chápať ako ďalšiu úroveň paralelnej organizácie populácie, ktorá ma za cieľ zvýšiť výkonnosť aj efektivitu genetických algoritmov. PGA prinášajú výhody v porovnaní s GA, ktoré sú: paralelné hľadanie vo veľkom mnohodimenzionálnom priestore potenciálnych riešení, lepšia možnosť vnútornej organizácie v porovnaní s GA, vyššia efektivita pri rovnakom počte vyhodnotení účelovej funkcie v porovnaní s GA, schopnosť nájsť lepšie riešenia, schopnosť uvolniť sa zo stagnácie v lokálnom extréme a smerovať ku globálnemu extrému.

Štruktúra topológie medzi populáciami môže byť rôzna. Jednotlivé populácie medzi sebou nemusia byť všetky previazané. Perióda migrácie by nemala byť veľmi krátka a migračné väzby by nemali byť príliš početné, pretože sa môže stať, že sa celý PGA bude správať ako keby bol jednou veľkou populáciou.

# 2 Grid

Jedným z cieľov tejto práce je preskúmať možnosti aplikovania útokov na klasické šifry v gridovom prostredí. Grid môžeme chápať ako skupinu počítačov, uzlov, spojenú pomocou siete Local Area Network (LAN), prípadne inou sieťovou technológiou. Jednotlivé uzly môžu, ale nemusia byť geograficky oddelené. Účelom takýchto počítačov je poskytnúť veľký výpočtový výkon, ktorý je použitý na riešenie špecifických úloh. Vrámci Slovenskej technickej univerzity (STU), Centra výpočtovej techniky (CVT) sa nachádza superpočítač IBM iDataPlex.

## 2.1 hpc.stuba.sk

Superpočítač IBM iDataPlex pozostáva z 52 výpočtových uzlov. Každý výpočtový uzol má nasledovnú konfiguráciu [5]:

- CPU: 2 x 6 jadrový Intel Xeon X5670 2.93GHz
- RAM: 48GB (24GB na procesor)
- HDD: 2TB 7200RPM SATA
- GPU: 2 x NVIDIA Tesla M2050 448 cuda jadier, 3GB ECC RAM
- Operačný systém: Scientific Linux 6.4
- Sieťové pripojenie: 2 x 10Gb/s Ethernet

Spolu máme k dispozícii 624 CPU, 3584 cuda jadier, 2,5TB RAM , 104TB lokálneho úložného priestoru a ďalších 115TB zdielaného úložiska. Výpočtový výkon dosahuje 6,76 TFLOPS a maximálny príkon aj spolu s chladením je 40kW [5].

V tabuľke 1 môžeme vidieť dostupné diskové umiestnenia pre každého používateľa, prípadne spustenú úlohu. Umiestnenie */home/$USER* je domovským priečinkom každého používateľa. Jedno z obmedzení tohto umiestnenia je, že môže obsahovať maximálne osemdesiattisíc súborov a priečinkov. Taktiež má značne obmedzenú kapacitu, čo sa nemusí hodiť pre každý typ úlohy. Ďalším umiestnením, ktoré ma používateľ k dispozícii je */work/$USER*. Toto umiestnenie nemá žiadne väčšie obmedzenia, slúži ako zdieľaný disk pre výpočty. Môžeme tu vytvárať ľubovolný počet súborov a priečinkov, avšak podľa [5] by sa tento disk mal využívať hlavne na prenos objemnejších dát v blokoch väčších ako 16kB. Obe spomenuté umiestnenia sú sieťové disky GPFS. Posledným umiestnením je */scratch/$PBS_JOBID* alebo tiež aj *$TMPDIR* v prípade PBS skriptu. Tento priestor

je unikátny pre každú úlohu, a je vhodný na spracovanie veľkého počtu malých súborov. V prípade použitia tohto umiestnenia si treba dať pozor na zmazanie dát, ktoré sa mažú ihneď po skončení úlohy.

| **Filesystem** | **Zálohovanie** | **Mazanie** | **Kapacita** | **Obmedzenia** |
|---|---|---|---|---|
| `/home/$USER` | áno | nie | 32GB | 80k inodes |
| `/scratch/$PBS_JOBID` | nie | ihneď | 1.6TB | nie |
| `/work/$USER` | nie | áno | 56TB | nie |

Tabuľka 1: Dostupné umiestnenia pre používateľa na hpc.stuba.sk

Aby sme boli schopní grid používať, musíme si najprv zaregistrovať projekt a požiadať o vytvorenie používateľského účtu na stránke výpočtového strediska `hpc.stuba.sk`. Po registrácii a získaní prihlasovacích údajov sa môžeme prihlásiť do webového rozhrania, cez ktoré môžeme spravovať projekt, pridávať ďalších riešiteľov, prezerať si štatistiky a grafy. Dôležitou funkciou webového rozhrania je zmena hesla používateľa a pridanie SSH verejného kľúča, pomocou ktorého sa prihlasujeme bez zadávania hesla.

## 2.2 Príkazy

Do gridu sa môžeme prihlásiť cez SSH zadaním príkazu *ssh <username>@login.hpc.stuba.sk* a následným zadaním hesla v prípade, ak nepoužívame prihlasovanie pomocou verejného kľúča. Ak sa pripájame mimo univerzitnej siete STU, na prihlásenie musíme použiť VPN. Po pripojení máme k dispozícii štandardnú linuxovú konzolu, ktorá ale obsahuje niekoľko špecifických príkazov pre daný grid. Zaujímať nás budú príkazy: *module, qstat, qfree, qsub, qsig*. Niektoré výstupy sú pre svoju obsiahlosť skrátené.

### 2.2.1 module

Príkaz *module* slúži na rýchle nastavenie ciest k vybraným knižniciam. Existujúce moduly môžeme vypísať pomocou *module avail*.

```
--------------------------- /apps/modulefiles -------------------------
abyss/1.3.7               gaussian/g03              mvapich2/2.1
ansys/15.0                gaussian/g09              mvapich2/2.2
cmake/2.8.10.2            gcc/4.7.4(default)        nwchem/6.1.1(default)
cmake/3.1.0               gcc/4.8.4                 nwchem/6.6
cp2k/2.5.1                gcc/4.9.3                 openblas/0.2.18
cuda/6.5                  gcc/5.4                   openmpi/1.10.2
devel                     gcc/6.3                   openmpi/1.10.4
dirac/13.3                gridMathematica/9.0       openmpi/1.10.5
```

```
dirac/14                  intel/composer_xe_2011    openmpi/1.4.5
esi/pamstamp              intel/composer_xe_2013    openmpi/1.6.5(default)
esi/pamstamp-platform     intel/libs_2011           openmpi/1.6.5-int8
esi/procast               intel/libs_2013           openmpi/1.7.2
esi/sysweld               matlab/R2015b             openmpi/1.7.5
fftw3/3.3.3               molcas/8.0                openmpi/1.8.8
fftw3/3.3.5               mvapich2/1.8a2            openmpi/1.8.8-int8
fftw3/intel-3.3.3         mvapich2/1.9(default)     openmpi/2.1.0
fluent/15.0.7             mvapich2/2.0              openmpi/intel-1.10.4
```

Výpis 1: module avail

Pre načítanie modulov zadáme *module load <modul1> <modul2> ...*, aktuálne používané moduly zobrazíme pomocou *module list* a odstrániť ich môžeme príkazom *module purge*. Podrobnejšie voľby príkazu *module* sa môžeme dozvedieť z manuálových stránok.

### 2.2.2 qstat

Ďalším dôležitým príkazom je *qstat*, ktorý zobrazuje status aktuálne bežiacich úloh. Detailnejší výpis o nami spustených úlohách môžeme vypísať cez *qstat -u $USER* alebo *qstat -a*.

```
Job ID         Name            User         Time Use  S Queue
----------     ------------    -----------  ---------  - --------
114557.one     halogen         3xjakubecj   499:03:0  R parallel
114640.one     JerMnchexFq5    3breza       218:35:9  R parallel
114663.one     Job4            3xrasova     78:07:20  R parallel
114668.one     run.opt         3antusek     674:08:1  R parallel
114692.one     Job5            3xbuchab     43:39:43  R parallel
114710.one     PGA             3xelias      226:46:1  R parallel
```

Výpis 2: qstat

```
Job ID     Queue    Jobname      SessID  TSK  Time      S   Time
---------- -------- ----------- ------ ---- --------- - --------
114710.one parallel PGA           3418   96   120:00:00 R 19:08:38
115265.one parallel PGA_Mpi_3_b  24619    4   120:00:00 R 31:17:51
115266.one parallel PGA_Mpi_3_d  14748    4   120:00:00 R 31:17:51
115267.one parallel PGA_Mpi_3_e  14780    4   120:00:00 R 31:17:51
115268.one parallel PGA_Mpi_5_b  16429    6   120:00:00 R 31:17:50
115269.one parallel PGA_Mpi_5_d  16471    6   120:00:00 R 31:17:49
115270.one parallel PGA_Mpi_5_e  22492    6   120:00:00 R 31:15:43
115271.one parallel PGA_Mpi_5_f  22450    6   120:00:00 R 31:15:45
115272.one parallel PGA_Mpi_11_b 4254   12   120:00:00 R 31:12:39
115273.one parallel PGA_Mpi_11_d  --    12   120:00:00 Q   --
```

```
115274.one parallel PGA_Mpi_11_e 1647  12  120:00:00 R 31:12:08
115275.one parallel PGA_Mpi_11_f 22605 12  120:00:00 R 31:11:37
```

Výpis 3: qstat -u 3xelias

Posledný riadok tabuľky príkazu *qstat -u 3xelias* popisuje nami spustenú úlohu. Dôležité sú pre nás predovšetkým stĺpce `Time`, `Job ID`. Posledný stĺpec `Time` hovorí o tom ako dlho je už naša úloha spustená, druhý stĺpec `Time` deklaruje maximálny možný čas, ktorý má úloha PGA vyhradený. Hodnoty zo stĺpca `Job ID` môžeme použiť do príkazu *qsig* pre vynútené ukončenie úlohy. Ukončiť úlohu príkazom *qsig* by sme mohli napríklad:

```
# ukoncenie ulohy
qsig 115275
```

### 2.2.3 qfree

Ak si chceme zobraziť aktuálne vyťaženie gridu, môžeme tak urobiť príkazom *qfree*.

```
CLUSTER STATE SUMMARY                Local   GPFS Storage

Core       1  ...  12  load  FreeMem Scratch Read  Write State
Node  Queue                    [GB]    [GB]   [MB/s] [MB/s]

comp01 S  [ ] ... [ ] 0.00 44.44  0 (0.0%) 0.00  0.00  free
comp02 T  [ ] ... [ ] 0.00 44.45  0 (0.0%) 0.00  0.00  free
...
comp44 P  [O] ... [O] 12.0 44.41  0 (0.0%) 0.00  0.00  job-exclusive
comp45 P  [O] ... [O] 12.0 44.40  0 (0.0%) 0.00  0.00  job-exclusive
comp46 P  [O] ... [O] 12.0 44.41  0 (0.0%) 0.00  0.00  job-exclusive
comp47 P  [O] ... [O] 12.0 44.41  0 (0.0%) 0.00  0.00  job-exclusive
comp48 P  [X] ... [X] 1.25 22.76  0 (0.0%) 45.50 2.83  job-exclusive
gpu1   G  [X] ... [X] 7.98 38.33  0 (0.0%) 42.13 0.92  job-exclusive
gpu2   G  [ ] ... [ ] 0.00 44.45  0 (0.0%) 0.00  0.00  free
gpu3   G  [ ] ... [ ] 0.00 44.45  0 (0.0%) 0.00  0.00  free
gpu4   G  [ ] ... [ ] 0.00 44.45  0 (0.0%) 0.00  0.00  free
```

Výpis 4: qfree

Prvý stĺpec popisuje názov výpočtového uzla. Druhý stĺpec označuje druh fronty. S je pre sériové úlohy, `T` pre interaktívne úlohy, podobne `P` pre paralelné výpočty a `G` pre grafické výpočty. Stĺpce jeden až dvanásť označujú procesory CPU, respektíve GPU pre grafické výpočty. Zvyšné stĺpce, ako už možno vyčítať z názvu, popisujú celkové zaťaženie výpočtového uzla, volnú pamäť a využitie diskov. Posledný stĺpec `State` popisuje stav uzla. Uzol môže byť voľný alebo vyťažený, ak vykonáva nejakú úlohu. Procesory, na ktorých

prebieha výpočet našej úlohy, sú označené ako `[0]`, zvyšné vyťažené CPU sú označené ako `[X]`, naopak voľné CPU sú označené medzerou `[ ]` a v prípade, ak uzol nie je dostupný, budú CPU označené ako `[-]`.

## 2.3 Výpočtové fronty

Posledný a najdôležitejší príkaz je *qsub*, slúži na zaradenie úloh, PBS skriptov do výpočtovej fronty. Aby sme boli schopní spustiť akúkoľvek výpočtovú úlohu na gride, potrebujeme k tomu Portable Batch System (PBS) súbor. PBS súbor je v skutočnosti iba jednoduchý textový súbor, ktorý definuje požiadavky na výpočtové zdroje a príkazy pre grid.

V tabuľke 2 sa nachádzajú všetky výpočtové fronty, ktoré sú dostupné na gride `hpc.stuba.sk`. Fronta *debug* slúži na rýchle odladenie úloh. Úlohy v tejto fronte majú vysokú prioritu a preto sú spustené takmer okamžite. *Debug* fronta je obmedzená na maximálne dve súčasne spustené úlohy. Fronta *gpu* je ďalším typom fronty pre úlohy, ktoré využívajú grafický akcelerátor. Pre úlohy, ktoré využívajú MPI, OpenMP a iné knižnice na paralelné programovanie, slúži fronta *parallel*. Úlohy takého typu musia použiť minimálne štyri a maximálne deväťdesiatšesť CPU. Poslednou výpočtovou frontou je *serial*. V tejto fronte môžeme spúšťať jednoprocesorové úlohy.

| **Názov fronty** | **walltime (max)** | **nodes** | **ppn** |
|---|---|---|---|
| debug | 30 minút | - | - |
| gpu | 24 hodín | - | - |
| parallel | 240 hodín | 1 - 8 | 4 - 12 |
| serial | 240 hodín | 1 | 1 |

Tabuľka 2: Výpočtové fronty a ich obmedzenia

### 2.3.1 Príklad sériovej úlohy

```
1 #!/bin/bash
2
3 #PBS -N uloha1
4 #PBS -l nodes=1:ppn=1
5 #PBS -l walltime=00:01:00
6 #PBS -A 3ANTAL-2016
7 #PBS -q serial
8
9 cd /work/3xelias/uloha1
```

```
10 ./seriova_uloha
```

Výpis 5: uloha1.pbs

Vo výpise 5 môžeme vidieť príklad jednoduchého skriptu sériovej úlohy. Poukážme na jednotlivé riadky skriptu:

- Prvý riadok v súbore definuje aký shell sa má použiť pre spustenie skriptu. V našom prípade sme použili BASH, ale mohli by sme použiť aj iný shell alebo skriptovací jazyk.
- Tretí riadok určuje názov úlohy.
- Štvrtý riadok vymedzuje, koľko uzlov a procesorov si žiadame od gridu. V tomto prípade si žiadame jeden výpočtový uzol a jeden procesor.
- Piaty riadok v PBS skripte vymedzuje aké časové rozpätie potrebujeme pre úlohu. V tomto príklade si žiadame jednu minútu.
- V šiestom riadku sa nachádza identifikátor, podľa ktorého sa identifikujú úlohy s jednotlivými projektami. Tento parameter je povinný a možno ho získať po prihlásení na webový portál `https://www.hpc.stuba.sk/index.php?l=sk&page=login`.
- Parameter *-q* v siedmom riadku definuje typ výpočtovej fronty do ktorej bude úloha zaradená. V tomto príklade chceme úlohu zaradiť do fronty *serial*.
- V deviatom riadku sa presunieme do priečinku, v ktorom sú uložené všetky potrebné dáta pre túto úlohu vrátane programu *seriova_uloha*.
- Posledný riadok spustí program *seriova_uloha*.

Úlohu môže zaradiť do fronty príkazom *qsub uloha1.pbs*.

### 2.3.2 Príklad paralelnej úlohy

```
1  #!/bin/bash
2
3  #PBS -N paralelna_uloha
4  #PBS -l nodes=5:ppn=12
5  #PBS -l walltime=48:00:00
6  #PBS -A 3ANTAL-2016
7  #PBS -q parallel
8  #PBS -m ea
9  #PBS -M xelias@stuba.sk
10
```

```
11 . /etc/profile.d/modules.sh
12 module purge
13 module load gcc/5.4 openmpi/1.10.2
14
15 cd /work/3xelias/parallel
16 mpirun ./parallel
```

Výpis 6: uloha2.pbs

Podobne ako v predchádzajúcom príklade sériovej úlohy si popíšeme niektoré riadky príkladu *uloha2.pbs*:

- Na rozdiel od predchádzajúcej úlohy, v tomto príklade si na riadku číslo štyri žiadame päť výpočtových uzlov a na každom uzle dvanásť CPU. Dokopy si žiadame šesťdesiat procesorov.
- V piatom riadku požadujeme časové rozpätie štyridsiatich ôsmich hodín.
- Siedmy riadok definuje paralelnú frontu.
- Parametre *e* a *a* na riadku osem hovoria o tom, kedy sa má poslať email o zmene stavu úlohy. Parameter *e* znamená po skončení úlohy. Parameter *a* znamená pri zrušení úlohy. Ďalšie parametre môžu byť *b* (štart úlohy) a *n* (neposielať žiadny e-mail) [6].
- V deviatom riadku definujeme e-mailovú adresu, na ktorú bude zaslaný mail v prípade, ak úloha skončí alebo bude prerušená.
- V jedenástom riadku načítame všetky potrebné premenné prostredia pre moduly.
- V dvanástom riadku odstránime všetky načítané moduly ak boli nejaké dostupné v premenných prostredia.
- V riadku číslo trinásť tohto súboru načítame knižnicu gcc verzie 5.4 a knižnicu openmpi verzie 1.10.2. Pre správny beh programu by sa všetky načítané knižnice mali zhodovať s tými, s ktorými bola aplikácia spúšťaná v tomto skripte skompilovaná.
- Podobne ako v ukážke 5 sa prepneme do priečinku */work/3xelias/parallel*, ktorý musí obsahovať všetky potrebné dáta pre samotný program *parallel*.
- Posledný riadok spustí program *mpirun*, ktorý potom spustí program *parallel*. Tento krok bude vysvetlený v nasledujúcej kapitole.

# 3 MPI

Message passing je forma programovania, ktorá sa používa pri paralelnom programovaní či už na viacjadrových procesoroch alebo v gridovom prostredí. V takejto forme programovania môže byť program rozdelený na viacero logických blokov alebo procesov. Takýchto procesov môže byť viac ako počet dostupných CPU, avšak zvyčajne by mal byť tento počet rovnaký. Procesy môžu vykonávať rozličné úlohy a môžu bežať na rozličných, geograficky oddelených CPU. [7]

Procesy medzi sebou komunikujú posielaním správ. Správy zvyčajne reprezentujú nejaké dáta, ale môžu slúžiť aj na synchronizáciu.

Táto forma programovania zaznamenala veľký rozmach hlavne v deväťdesiatych rokoch minulého storočia, keď takmer každý predajca paralelných systémov ponúkal vlastnú implementáciu message passing prostredia.

Následkom týchto udalostí vzniklo Message Passing Interface (MPI) Forum, čo bola skupina viac ako osemdesiat ľudí zo štyridsiatich rôznych organizácií, ktoré predstavovali predajcov paralelných systémov, používateľov ale aj výskumné laboratóriá a univerzity. [7]

Úlohou MPI fóra bolo zjednotiť message passing systémy a navrhnúť nový systém. Systém by mal podporovať komplexné dátové štruktúry, bezpečnú komunikáciu a byť dostatočne modulárny.

Výsledkom práce MPI fóra v júni roku 1994 bol vznik štandardu MPI-1.0, ktorý je dodnes akceptovaný a má mnoho používateľov aj napriek tomu, že už existujú novšie verzie štandardu. MPI štandard nie je knižnica. Je to špecifikácia toho, ako by mala konkrétna implementácia knižnice vyzerať. Existuje viacero implementácií. Medzi najznámejšie patria OpenMPI, MVAPICH, IBM MPI.

Aj napriek tomu, že MPI štandard je veľmi rozsiahly, obsahuje stovky funkcií, zaujímať sa však budeme len o niektoré z nich.

## 3.1 Point-to-point komunikácia

Odosielanie a prijímanie správ je kľúčovým stavebným mechanizmom MPI komunikácie. Základné operácie sú *send* (posielanie) a *receive* (prijímanie). Výmene správ medzi dvoma procesmi hovoríme *point-to-point* komunikácia. Takmer všetky MPI konštrukcie sú založené na point-to-point komunikácii [8]. Pre ilustráciu si uveďme jednoduchý príklad:

```
1 #include <stdio.h>
2 #include <mpi.h>
3 int main(int argc, char **argv) {
```

```
4       char msg[20];
5       int rank;
6       MPI_Init(&argc, &argv);
7       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8       if (rank == 0) {
9           strncpy(msg, "Hello, MPI", 20);
10          MPI_Send(msg, strlen(msg) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
11      } else if (rank == 1) {
12          MPI_Recv(msg, sizeof(msg), MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
13          printf("%s\n", msg);
14      }
15      MPI_Finalize();
16      return 0;
17  }
```

Výpis 7: Point-to-point komunikácia

V tomto príklade proces nula *(rank == 0)* pošle správu procesu prostredníctvom send operácie *MPI_SEND*. Táto operácia definuje takzvaný *send buffer*. *Send buffer* je miesto v pamäti odosielateľa, v ktorom sa nachádzajú dáta na odoslanie. *Send buffer* je v tomto prípade premenná *msg* nachádzajúca sa v pamäti procesu nula. Prvé tri argumenty *send* operácie špecifikujú dáta pre príjemcu. Táto správa bude obsahovať pole charov, respektíve string. Ďalšie tri parametre *send* operácie definujú prijímateľa správy. Proces jedna *(rank == 1)*, prijme správu s *receive* operáciou *MPI_Recv*. Správa je prijatá na základe zadaných parametrov a dáta sú uložené do *receive buffera*. V tomto príklade je *receive buffer* premenná *msg* v pamäťovom priestore procesu jedna. Prvé tri parametre definujú dáta, ktoré chce príjemca prijať. Ďalšie tri parametre slúžia na zvolenie správy od odosielateľa. Posledný parameter sa používa na získanie informácií o prijatej správe. V tomto príklade tento parameter ignorujeme.

Program z výpisu 7 môžeme skompilovať a následne spustiť v dvoch krokoch:

```
mpicc main.c -o p2p
mpirun -n 2 ./p2p
```

Výsledkom programu a komunikácie týchto dvoch procesov bude, že proces jedna vypíše *Hello, MPI*. Program *mpicc* slúži ako wrapper pre kompilátor *gcc*. Jeho úlohou je linkovanie knižnice *mpi*. Program *mpirun* slúži na spúšťanie *mpi* programov.

## 3.2 Blokujúca komunikácia

Obe MPI volania vo výpise 7 boli blokujúce. To znamená, že odosielateľ čaká na volaní *MPI_Send* dovtedy, kým príjemca správu prijme. To isté platí aj pre príjemcu,

ktorý čaká na volaní funkcie *MPI_Recv* až kým nedostane správu od odosielateľa.

```
MPI_Send(
const void* buf,
int count,
MPI_Datatype datatype,
int dest,
int tag,
MPI_Comm comm)
```

Výpis 8: MPI_Send

Vo výpise 8 vidíme definíciu volania *MPI_Send*. Prvým argumentom funkcie je *adresa receive buffera*, ďalší argument je dĺžka buffera (*count*), ktorá musí byť celé kladné číslo. V prípade, že táto podmienka nie je splnená MPI volanie vráti chybu. Typ buffera špecifikujeme pomocou argumentu *datatype*. Niektoré základné dátové typy môžeme vidieť v tabuľke 3. Ďalšie tri argumenty slúžia na definovanie príjemcu. Pomocou parametra *dest* môžeme zvoliť komu je správa adresovaná. V prípade, že si odosielateľ a príjemca vymieňajú viac správ rôzneho typu alebo rôzneho obsahu, potrebujeme tieto správy nejako odlíšiť. Na odlíšenie správ slúži argument *tag*. *Tag* musí byť kladné celé číslo. V príklade výpisu 7 sme ako *tag* použili nulu, avšak mohli sme použiť aj inú hodnotu. Posledným argumentom je *comm*, ktorý slúži na definovanie komunikačnej skupiny. Základná komunikačná skupina je svet (*MPI_COMM_WORLD*), kde si všetci môžu vymieňať správy. MPI štandard zahŕňa aj vytváranie vlastných podskupín komunikátorov, napríklad pomocou volania *MPI_Comm_create_group* [8].

```
MPI_Recv(
const void* buf,
int count,
MPI_Datatype datatype,
int source,
int tag,
MPI_Comm comm,
MPI_Status *status)
```

Výpis 9: MPI_Recv

MPI volanie *MPI_Recv* používa veľmi podobnú syntax ako *MPI_Send*. Prvé tri argumenty sú identické. Argument *source*, v prípade funkcie *MPI_Recv*, definuje rank odosielateľa. Ďalším rozdielom je argument *status*, ktorý sa používa na získanie informácií o prijatej správe, alebo môže byť aj ignorovaný ako v príklade výpisu 7. Štruktúra *MPI_Status* obsahuje 3 hlavné informácie: rank odosielateľa, tag správy a dĺžku správy.

| **MPI_Datatype** | **C ekvivalent** |
|---|---|
| MPI_SHORT | short int |
| MPI_INT | int |
| MPI_LONG | long int |
| MPI_LONG_LONG | long long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_UNSIGNED_LONG_LONG | unsigned long long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | char |

Tabuľka 3: Dátové typy v MPI a ich C ekvivalenty

## 3.3 Neblokujúca komunikácia

Výkonnosť programov môže byť v mnohých prípadoch vylepšená prekrývaním blokujúcich volaní. Jedným z možných spôsobov ako toto dosiahnuť je použitie threadov . Alternatívny spôsob ako zlepšiť výkon programu môže byť použitie neblokujúcich volaní. [8]

Pri neblokujúcej komunikácii si môžeme zadefinovať štyri rôzne neblokujúce operácie: *send* pre posielanie, *recv* pre prijímanie, *send complete* pre dokončenie posielania a *recv complete* pre dokončenie prijatia. Neblokujúce volanie *send* spustí *send* operáciu, ale nedokončí ju. Toto volanie inicializuje skopírovanie do *send buffera* ale samotné kopírovanie nedokončí. Na dokončenie operácie *send* je potrebné ďalšie volanie *send complete*, ktoré overí, že dáta boli skutočne skopírované a prenesené príjemcovi. Neblokujúca *send* operácia zvyčajne beží paralelne s vykonávaným programom. Podobne aj neblokujúce volanie *recv* inicializuje *recv* operáciu, ale nedokončí ju. Na dokončenie potrebuje separátne volanie *recv complete*, ktoré zaručí, že dáta boli prenesené do *recv buffera*. Neblokujúca *recv* operácia podobne ako *send* pokračuje paralelne so zvyškom programu.

```
1 MPI_Request request[2];
2 if (rank == 0) {
3     char msg1[] = "Hello, ", msg2[] = "MPI";
```

```
4       MPI_Isend(msg1, strlen(msg1) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &request[0]);
5       MPI_Isend(msg2, strlen(msg2) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &request[1]);
6       zlozity_vypocet();
7       MPI_Waitall(2, request, MPI_STATUSES_IGNORE);
8
9   } else if (rank == 1) {
10      char msg1[20], msg2[20];
11      MPI_Irecv(msg1, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request[0]);
12      MPI_Irecv(msg2, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request[1]);
13      iny_vypocet();
14      MPI_Waitall(2, request, MPI_STATUSES_IGNORE);
15      printf("%s%s\n", msg1, msg2);
16  }
```

Výpis 10: Neblokujúca komunikácia

Vo výpise 10 môžeme vidieť príklad neblokujúcej komunikácie. V tomto príklade ako *send* operácia slúži funkcia *MPI_Isend*. Proces nula *(rank == 0)* pošle najprv správu *msg1* a následne správu *msg2*. Obe volania sú neblokujúce, to znamená, že program nečaká a pokračuje funkciou *zlozity_vypocet*. Po dokončení výpočtu čaká na doručenie správ. Tento krok je dôležitý, pretože ak by sme nečakali na dokončenie prenosu správy, premenné *msg1* a *msg2* by už viacej nemuseli byť validné, za predpokladu že vyjdeme z if konštrukcie. V prípade procesu jedna *(rank == 1)* príjemca inicializuje neblokujúce volanie *MPI_Irecv* a pokračuje funkciou *iny_vypocet*. Po dokončení výpočtu skontroluje či už boli dáta prijaté volaním *MPI_Waitall*. Na tomto volaní čaká v prípade ak dáta ešte neboli prijaté. Tento krok je pre príjemcu dôležitý, pretože mu zaručuje, že dáta boli prijaté a môže ich použiť.

Výstupom tohto programu môže byť „*Hello, MPI*“ alebo aj „*MPIHello,* “, pretože poradie vykonania operácií nie je zaručené. Správne poradie správ by sme docielili rôznymi *tagmi* pre správy *msg1* a *msg2*.

## 3.4 Dynamická alokácia

Vo výpise 7 sme použili staticky alokovanú pamäť, čo je vo väčšine prípadov nepraktické a často môže viesť k chybám. Ak by v tomto príklade odosielateľ poslal dlhšiu správu ako je príjemca schopný prijať, tak by tento program zlyhal.

Riešením tohto problému je dynamická alokácia pamäte pre prijaté správy. Aby sme mohli pamäť dynamicky alokovať, potrebujeme vedieť koľko dát sa nám snaží odosielateľ poslať. To môžeme zistiť pomocou štruktúry *MPI_Status* a volaní *MPI_Probe, MPI_Get_count*. Kód procesu jedna z príkladu 7 by sme mohli nahradiť nasledovne:

```
MPI_Status status;
```

```
int msg_size;
MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_CHAR, &msg_size);
char *buf = (char *)malloc(msg_size);
MPI_Recv(buf, msg_size, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("%s\n", buf);
free(buf);
```

## 3.5 Serializácia dátových typov

Častokrát si procesy pri MPI komunikácii medzi sebou potrebujú vymieňať nie len primitívne dátové typy, ale aj zložitejšie dátové štruktúry. Odosielateľ musí dátové štruktúry pred komunikáciou rozložiť na primitívne typy. Spôsob ako sa dáta rozložia je medzi odosielateľom a príjemcom vopred dohodnutý. Príjemca po prijatí dát znovuposkladá dátové štruktúry do pôvodného stavu. Tomuto procesu sa hovorí serializácia dát.

MPI rieši tento problém napríklad volaniami *MPI_Type_create_struct, MPI_Pack*. Avšak tieto volania častokrát vyžadujú príliš nízkoúrovňové programovanie.

# 4 Návrh a implementácia

Jedným zo základných cieľov tejto práce je poskytnúť jednoduchšie vývojové prostredie pre ďalších lúštiteľov a programátorov, ktorí budú pri svojej práci využívať grid. Takéto prostredie by malo poskytnúť spoločné stavebné bloky, ktoré by si riešitelia medzi sebou mohli navzájom zdieľať. Stavebné bloky by mohli zahŕňať napríklad implementácie šifier, ohodnocovacích funkcií alebo prácu so súbormi. Ďalšou spoločnou črtou je vytváranie projektov, ktoré by malo byť čo najviac automatické a malo by generovať štruktúru priečinkov a základných súborov.

Keďže vývoj priamo na gride v konzolovom prostredí je nepraktický, vývoj v nami navrhovanom prostredí by mal spĺňať nasledovné základné požiadavky:

- Automatické vytvorenie projektu.
- Lokálny vývoj a testovanie.
- Zdieľanie modulov medzi projektami.
- Automatické generovanie a úprava skriptov pre gridové prostredie.
- Jednoduchá synchronizácia s gridom hpc.stuba.sk.

## 4.1 Štruktúra prostredia

V tejto kapitole si definujeme štruktúru prostredia a význam jednotlivých priečinkov a súborov. Toto prostredie obsahuje skripty a zdrojové kódy potrebné pre vývoj v lokálnom prostredí ale aj na gride.

Na nasledujúcom výpise môžeme vidieť štruktúru navrhnutého prostredia:

```
.
|- build.sh
|- CMakeLists.txt
|- module/
|   |- CMakeLists.txt
|   |- modul1/
|   |   |- CMakeLists.txt
|   |   |- ... h/cpp
|   |   |- ... h/cpp
|   |- modul2/
|   |- ...
|- project/
|   |- CMakeLists.txt
```

```
|   |- projekt1/
|   |   |- CMakeLists.txt
|   |   |- projekt1
|   |   |- projekt1.pbs
|   |   |- src/
|   |   |-  |- main.cpp
|   |   |-  |- h/cpp
|   |   |-  |- ...
|   |- projekt2/
|   |-  |- ...
|   |- ...
|- vendor/
    |- lib1
    |- lib2
    |- ...
```

Výpis 11: Štruktúra prostredia

Skript *build.sh* vytvára projekty do podpriečinku *project*. Každý projekt po vytvorení obsahuje automaticky generované súbory: *CMakeLists.txt*, binárny spustiteľný súbor (po kompilácii), *.pbs* súbor a *main.cpp* v priečinku *src*. Do priečinku *src* by sa mali umiestňovať zdrojové súbory špecifické pre daný projekt. Ostatné zdrojové kódy, ktoré nesúvisia s daným projektom by mali ísť do modulov (priečinok *module*). Priečinok *module* slúži na zdieľanie zdrojových kódov (modulov) medzi projektami. Posledným priečinkom je *vendor*, ktorý slúži na uchovávanie knižníc od tretích strán.

## 4.2 Technológie

V tejto práci sme si zvolili ako programovací jazyk *C++*, ktorý je dostatočne rýchly a poskytuje bohatú štandardnú knižnicu. Tento programovací jazyk sa taktiež vyučuje na Fakulte elektrotechniky a informatiky a je veľmi rozšírený v industriálnej sfére.

Keďže MPI štandard od verzie 3.0 nepodporuje *C++* volania [8], budeme používať knižnicu Boost a jej MPI nadstavbu. Z Boost knižnice tiež využijeme modul serialization pre dátovú serializáciu.

Vrámci gridu hpc.stuba.sk nemáme k dispozícii knižnicu boost. Tiež nemáme možnosť inštalovať žiadne ďalšie knižnice, a preto sa pred prvou kompiláciou knižnica Boost stiahne a automaticky skompiluje pre dané prostredie. Kompilácia prebieha iba jedenkrát a knižnica sa uloží do priečinku *vendor/boost_<verzia>*.

Ako buildovací systém sme použili cmake. Cmake je multiplatformový, open-source nástroj, ktorý sa používa na buildovanie a testovanie softvéru [9]. Používa jednoduché kon-

figuračné súbory CMakeLists.txt, ktorých úlohou je generovanie štandarných buildovacích súbor [2]. Tento nástroj sme si tiež zvolili pretože dokáže vygenerovať naivné prostredie na kompiláciu zdrojového kódu a knižníc. Taktiež záhŕňa podporu viacerých „projektov“ vrámci jedného repozitáru. Ďalšou výhodou je, že môže byť použitý v kombinácii s takmer ľubovolným grafickým vývojovým prostredím.

Na správu zdrojových kódov sme použili systém *git*, pretože archiváciu a správu zdrojových kódov považujeme za veľmi dôležitú. Zdrojové kódy tohto projektu sú dostupné v prílohe tejto práce, ale tiež na githube `https://github.com/melias122/grid`.

Okrem archivácie zdrojových kódov považujeme za dôležité aj to, ako je zdrojový kód naformátovaný. Na formátovanie zdrojových súborov sme použili nástroj *clang-format*.

## 4.3 Skript build.sh

Základným príkazom celého prostredia je skript *build.sh* v kombinácii s buildovacím nástrojom cmake. Skript *build.sh* sa používa na vytváranie nových projektov, kompilovanie, synchronizáciu s gridom a na spúšťanie projektov v lokálnom prostredí, ale aj na gride. Základné operácie tohto príkazu *build.sh* sú:

- `-b/--build`, tento argument skompiluje všetky projekty a moduly nachádzajúce sa v priečinkoch *project*, *module* do priečinku *.build*. Taktiež skompiluje všetky potrebné závislosti a sformátuje zdrojové kódy, v prípade existencie nástroja *clang-format*. Tento mód kombinuje príkazy: *module* (kap. 2.2.1), *clang-format*, *cmake* a *make*.

- `-h/--help` argument vypíše návod ako pracovať s týmto príkazom.

- `-r/--run [projekt]` spustí projekt, ktorý je zadaný ako ďalší argument tohto príkazu. Názov projektu sa musí zhodovať s niektorým vytvoreným projektom z priečinku *project*. Hlavou myšlienkou tohto arugmentu je, že by mal zjednocovať spúštanie projektov rovnako v lokálnom prostredí ale aj na gride. Príkaz v tomto móde parsuje *main.cpp* v priečinku projektu, z ktorého sa snaží získať nastavenia *nodes*, *ppn*, *walltime* a *queue*. Na základe týchto nastavení pred každým spustením upraví *.pbs* súbor. V prípade, že používame tento mód v lokálnom prostredí, príkaz použije nastavenia ako argumenty pre *mpirun* na simulovanie gridového prostredia.

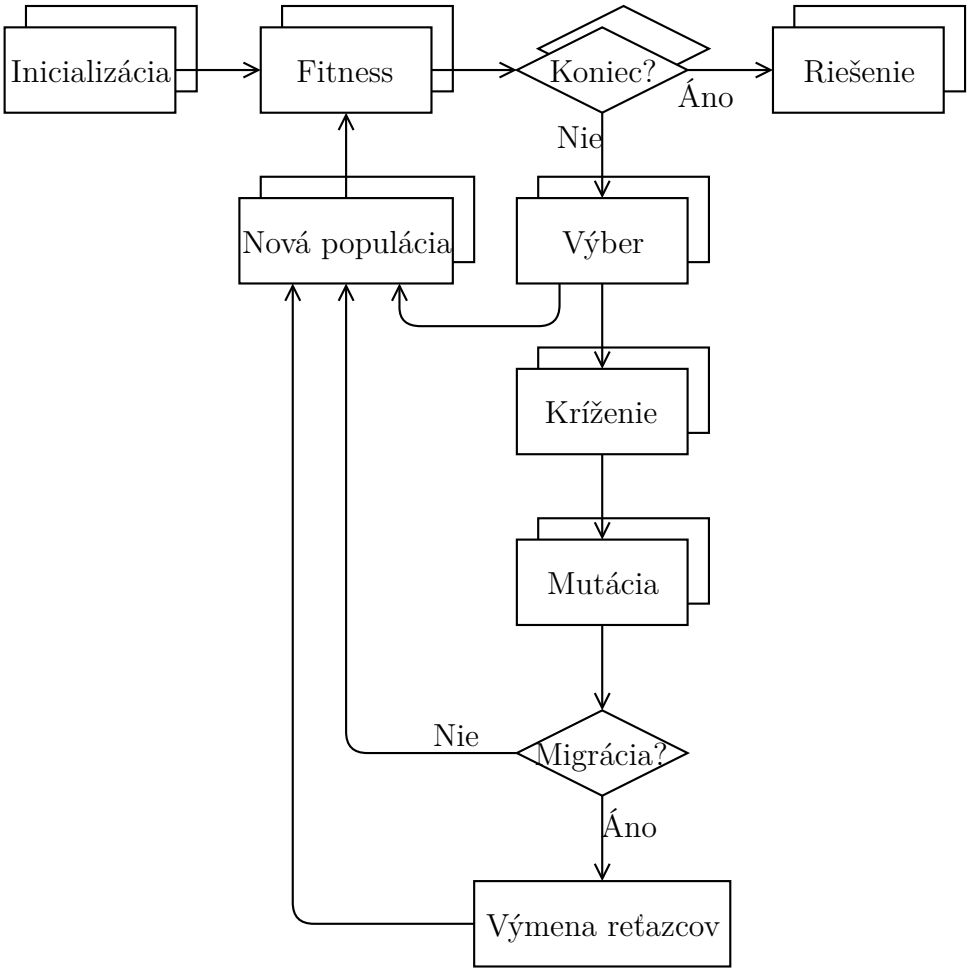
- `-n/--new-project [projekt]` vytvorí projekt do priečinku *project*. Tento príkaz automaticky vytvára základné súbory pre projekt, ktorými sú: *main.cpp*, a *.pbs* súbor, *CMakeLists.txt*, *.gitignore*.

[2] Makefile pre Unix, workspaces pre Windows MSVC

- `--sync` zosynchronizuje lokálny projekt na grid do priečinku */work/$USER/grid*. Priečinok work bol zvolený narozdiel od *home*, pretože nekladie žiadne obmedzenia na počet súborov. Tento príkaz nemá deštruktívny charakter, nezmazáva žiadne súbory. Na použitie tohto módu je potrebné mať vytvorený súbor *.username*, ktorý musí obsahovať login do gridu. V prípade, že tento súbor neexistuje, príkaz *build.sh* v tomto móde upozorní používateľa aby si ho vytvoril.

## 4.4 PGA modul

V tejto kapitole si ukážeme implementačné časti PGA modulu, ktorý sme integrovali a rozširovali v tomto vývojovom prostredí. Základnú implementáciu GA poskytol vedúci práce Ing. Eugen Antal, PhD. Táto implementácia bola ďalej nami rozvíjaná a do značnej miery upravená, niektoré časti kódu boli prepísané, avšak základná idea ostala zachovaná.




Obrázok 2: Štruktúra PGA s migračným modelom (vytvorené na základe [1])

Na obrázku 2 môžeme vidieť zjednodušenú blokovú schému PGA s migračným mode-

lom. Pri takomto type prebieha výpočet jednotlivých genetických algoritmov vo viacero nezávislých, ale spolupracujúcich subpopuláciach (ostrovoch). Tieto ostrovy môžu bežať na viacerých procesoroch alebo aj nezávislých počítačoch. Ostrovy si medzi sebou po určitom počte generácií a na základe vopred určených väzieb (topológií) vymenia niektorých jedincov, napríklad najlepších v aktuálnej generácii. Potom pokračujú izolovane ďalej. Procesu výmeny jedincov medzi jednotlivými GA hovoríme migrácia.

| **Operácia** | **Typ** | **Popis** |
|---|---|---|
| **Select** | Elitism(n) | Vyberie n najlepších jedincov. |
| | Random(n) | Náhodne zvolí n jedincov. |
| | Tournament(n) | N-krát zvolí 2 jedincov a vyberie lepšieho. |
| | Worst(n) | Vyberie n najhších jedincov. |
| **Crossover** | SinglePoint(1 \|\| 2) | Zvolí nádhoný bod a skríži 2 rodičov. Vráti 1 alebo 2 potomkov. |
| | Uniform(1 \|\| 2) | Zvolí niekoľko náhodných bodov a skríži 2 rodičov. Vráti 1 alebo 2 potomkov. |
| **Mutation** | Swap(n) | N-krát vymení 2 rôzne gény. |
| | SwapAll | Zmení všetky gény jedinca. |
| | WithProbability(p) | S pravdepodobnosťou p zmení náhodný gén. |

Tabuľka 4: Operácie GA

Základné stavebné bloky GA v našej implementácii sú:

**Genes** Obsahuje jednotlivé gény a predtavuje kľúč šifry. Z implentačného hľadiska sme zvolili reprezentáciu pomocou *std::string*. Výhodou tejto reprezentácie je jednoduchosť a aj o niečo lepší výkon, napríklad oproti *std::vector*.

**Chromosome** Je tvorený z génov a uchováva hodnotu skóre. Ohodnocovanie génov prebieha v našom modeli pomocou fitness funkcie a aj šifrovacej funkcie.

**Generator** Generuje gény pre chromozóm. Gény sú generované z abecedy podľa vopred určeného modelu. Generator je interface a teda umožňuje rôzne implementácie, pre rôzne reprezentácie génov. Pre monoalfabetickú substitúciu sme napríklad vytvorili *ShuffleGenerator*, ktorý generuje náhodné permutácie zo zadanej abecedy.

**Population** Obsahuje chromozómy a je reprezentovaná pomocou *std::vector*.

**Cipher** Interface, ktorý definuje šifru pomocou dešifrovacej funkcie.

**Fitness** Interface, ktorý definuje vyhodnocovaciu funkciu.

**Migrator** Abstraktná trieda, ktorá definuje migračnú logiku a vytvára topológie.

**Select** Abstraktná trieda, ktorá definuje operáciu výber.

**GeneticOperation** Abstraktná trieda, definuje operácie kríženie a mutácia.

**Schema** Popisuje genetický algoritmus. Obsahuje počet iterácií, veľkosť počiatočnej populácie, generátor, lúštenú šifru, fitness funkciu a migrátor v prípade, že používame PGA. Schéma tiež umožňuje pridávať operácie. Operácia je dvojica zložená z výberu (*Select*) a genetickej operácie (*Mutation*, *Crossover*). Dostupné operácie môžeme vidieť v tabuľke 4.

### 4.4.1 Genetický algoritmus

Genetický algoritmus v module PGA je implementovaný nasledovne:

```
1  Population GeneticAlgorithm::run(int id, const Scheme &scheme) {
2      Population pop;
3      for (int i = 0; i < scheme.initialPopulation; i++) {
4          pop.emplace_back(scheme.generator, scheme.cipher, scheme.fitness);
5      }
6      for (int i = 1; i <= scheme.iterations; i++) {
7          if (scheme.migrator) {
8              scheme.migrator->migrate(id, i, pop);
9          }
10         Population newpop;
11         for (const std::vector<Operation> &ops : scheme.operations) {
12             Population subpop = pop;
13             for (const Operation &op : ops) {
14
15                 subpop = op.select->select(subpop);
16                 op.geneticOperation->apply(subpop);
17
18                 for (Chromosome &c : subpop) {
19                     c.calculateScore(scheme.cipher, scheme.fitness, scheme.cache);
20                 }
21             }
22             append(newpop, subpop);
23         }
24         pop.swap(newpop);
25     }
26     std::sort(pop.begin(), pop.end(), Chromosome::byBestScore());
27     return pop;
28 }
```

Výpis 12: Genetický algoritmus

Základom tohto GA je trieda *Scheme*, ktorá definuje štruktúru genetického algoritmu. V prípade, že používame migračný model, algroritmus musí mať unikátne *id*. Na riadku 2 až 5 algoritmus vytvorí jednotlivé chromozómy populácie veľkosti *initialPopulation*. Gény

výtvára generátor a skóre je vypočítané pomocou šifry a fitness funkcie. Ak je migrátor definovaný v schéme, tak na riadku 7-9 vykonáme migráciu populácie. Migračná logika sa odohráva vo vnútri migrátora a migrovaní jedinci sa pridajú do aktuálnej populácie. Následne sa na aktuálnu populáciu na riadkoch 10 až 21 budú vykonávať operácie podľa vytvorenej schémy. Operácie môžu mať aj podoperácie a vytvárajú tak akúsi reťaz operácií. Na riadku 12 sa vytvorí kópia pôvodnej populácie, na ktorú sa aplikuje reťaz operácii. Riadok 13 prechádza cez jednotlivé operácie. Vždy sa vykoná najprv výber (*Select*) a na vybratú subpopuláciu je aplikovaná genetická operácia kríženia alebo mutácie (riadok 15 a 16). Tejto subpopulácii sa vyráta skóre. V tomto modeli musíme rátať skóre pri každej subpopulácii, pretože niektorá operácia, napríklad *Swap*, mohla zmeniť gény, a teda aj skóre jedinca, ktoré by mohlo byť použité v ďalšom výbere. Riadok 22 zjednocuje jednotlivé subpopulácie a na riadku 24 sa aktuálna populácia vymení za novú. Po prejdení všetkých iterácií už len algoritmus zoradí jedincov podľa najlepšieho skóre a vráti výslednú populáciu.

Pomocou triedy *Scheme* by sme mohli vytvoriť jednoduchú schému GA, napríklad takto:

```
1 ShuffleGenerator gen(alpha);
2 Monoalphabetic cipher(ciphertext);
3 L1DistanceBigrams fitness(bigrams);
4 Scheme s(10000, 30, &gen, &cipher, &fitness);
5 s.addOperations(new Select::Elitism(1), new Mutation::Swap(1));
6 auto subop = s.addOperations(new Select::Elitism(2), new Crossover::SinglePoint(2));
7 subop.emplace_back(new Select::Interface, new Mutation::WithProbability(0.25));
8 ... // dalsie operacie
```

V tomto príklade by sme vytvorili *ShuffleGenerator* z danej abecedy *alpha*; monoalfabetickú šifru s *ciphertextom*, ktorý chceme lúštiť; bigramovú fitness funkciu s referenčnými hodnotami *bigrams*. Schéma s má *10000* iterácií GA, počiatočná populácia má veľkosť *30* jedincov. Schéma obsahuje dve zreťazenia operácií. Prvé zreťazenie je na riadku 5, kde sa vyberie jeden elitný jedinec, na ktorého je aplikovaná operácia *Swap*. Druhé zreťazenie je na riadku 6 a 7, kde sa z populácie vyberú dvaja elitní jedinci, ktorí sa skrížia a vytvoria dvoch potomkov. Na riadku 7 potom pridáme suboperáciu. Výber len skopíruje populáciu, na ktorú sa aplikuje mutácia. Jednotlivých podoperácií ako aj zreťazení môžeme vytvárať ľubovolne veľa.

### 4.4.2 Serializácia chromozómov

Ako už bolo spomenuté v kapitole 3.5, pri MPI komunikácii častokrát potrebujeme vymienať aj zložené dátové typy ako štrukúry, triedy, vektory a iné. Pri implementácii

PGA migrátora sme narazili na tento problém. Keď nastane čas migrácie, jednotlivé ostrovy si medzi sebou potrebujú vymeniť časť populácie. Aby migrátor mohol posielať a aj prijímať vektory chromozómov (populácie). Do triedy *Chromosome* sme museli implementovať serializačnú procedúru, čo bolo vďaka použitiu Boost knižnice veľmi jednoduché.

Na nasledujúcich riadkoch môžeme vidieť ako vyzerala trieda *Choromosome* pred úpravou potrebnou na serializáciu:

```
1  class Chromosome {
2  public:
3    // public methods ...
4  private:
5      Genes m_genes; // std::string
6      double m_score;
7  };
8  using Population = std::vector<Chromosome>;
```

Ďalše riadky zobrazujú triedu *Chromosome* po úprave na serializáciu aj s potrebnými include súbormi z knižnice Boost.

```
1  #include <boost/serialization/access.hpp>
2  #include <boost/serialization/string.hpp>
3  #include <boost/serialization/vector.hpp>
4
5  class Chromosome {
6  public:
7    // public methods ...
8  private:
9      friend class boost::serialization::access;
10     template <class Archive>
11     void serialize(Archive &ar, const unsigned int version) {
12         ar &m_genes;
13         ar &m_score;
14     }
15     Genes m_genes; // std::string
16     double m_score;
17 };
18 using Population = std::vector<Chromosome>;
```

Hlavnou myšlienkou tohto riešenia je sprístupnenie privátnych premenných a funkcie serialize triedy *Chromosome*. To sme dosiahli pridaním riadku 9. Na riadku 10 až 15 je uvedený kód serializácie, ktorý je veľmi jednoduchý aj pre elegantnosť Boost knižnice, v ktorej je už serializácia pre stringy, vektory aj primitívne typy implementovaná.

### 4.4.3 Mpi migrátor a topológie

Aby sme boli schopní vytvoriť PGA, potrebujeme okrem genetického algoritmu nejako vytvárať topológie, ktoré medzi sebou dokážu komunikovať.

Vytváranie topológií sa nám podarilo zovšeobecniť nasledovným spôsobom:

```
1 class Migrator {
2     public:
3     Migrator(int migrationTime);
4     virtual void migrate(int senderId, int itteration, Population &population) = 0;
5     // sender -> receiver
6     void addMigration(int senderId, int receiverId, Migration::Type t, int n)
7     // ...
8 };
```

Výpis 13: Abstraktná trieda Migrator

*Migrator* je abstraktná trieda, ktorá definuje interface *migrate* a pomocnú funkciu *addMigration* na vytváranie topológií. Ideou tohto modelu je, že každý GA migruje v rovnakom čase. Funkcia *addMigration* vytvára migračné väzby typu odosielateľ (*senderId*) posiela príjemcovi (*receiverId*) *n* jedincov typu *t*. Pričom typ t môže byť napríklad: najlepší, najhorší, náhodný a iné. Pre príjemcu samozrejme platí, že očakáva migráciu rovnakého typu. Interface *migrate* musí byť implementovaný pre konkrétny PGA *Migrator*. Konkrétne implementácie môžu zahŕňať napríklad: thready, MPI, OpenMP, a ďalšie. Pomocou Migrator-a by sme mohli vytvoriť napríklad topológiu f z obrázku 10, ktorá by obsahovala 5 ostrovov.

```
1 int size = 5;
2 Migrator m;
3 for (int i = 0; i <= size - 2; i += 2) {
4     m.addMigration(i + 1, 0, Migration::Type::Best, 1);
5     for (int j = 1; j <= size - 2; j += 2) {
6         m.addMigration(i + 2, j, Migration::Type::Best, 1);
7     }
8 }
```

V tejto práci sme implementovali MPI migrátor nasledovným spôsobom:

```
1 void MpiMigrator::migrate(int senderId, int itteration, Population &population)
2 {
3     // nonblocking send
4     for (const Migration &m : p_senderMigrations[senderId]) {
5         Population pop = m.select(p);
6         mpi::request r = comm.isend<Population>(m.receiverId(), 0, pop);
```

```
7       }
8       // recv
9       for (const int &id : p_receiverMigrations[senderId]) {
10          Population pop;
11          comm.recv<Population>(id, 0, pop);
12          append(p, pop);
13      }
14      // wait for non blocking send
15      for (auto &r : request) {
16          r.wait();
17      }
18  }
```

Výpis 14: Kód MpiMigrator-a

*MpiMigrator* pošle niektorých jedincov populácie vopred zadaného typu podľa vytvorenej migračnej schémy (riadok 4-7). Trieda *Migration* na riadku 4 je pomocná trieda, ktorá v sebe uchováva väzby medzi jednotlivými uzlami a uskutočňuje výber jedincov. Posielanie v tomto prípade nesmie byť blokujúce, pretože by sa každý odosielateľ snažil iba posielať, ale nikto by nič neprijímal, čo by znamenalo deadlock. Naopak prijímanie blokujúce byť môže (riadok 9-13). Migrátor príjme nových jedincov ak existuje k nemu priradená väzba. Prijatých jedincov potom zaradí do aktuálnej populácie a nakoniec čaká na dokončenie neblokujúcich volaní (riadok 15-17).

## 4.5 Príklad použitia

Táto kapitola by mala slúžiť ako manuál pre ďalších používateľov. Postupne si prejdeme všetky kroky potrebné k použitiu nami vytvoreného softvéru od počiatočných nastavení prostredia, vývoja v lokálnom prostredí, až po spustenie na gride.

### 4.5.1 Lokálny vývoj

Na lokálny vývoj sme použili linuxový operačný systém *Ubuntu 16.04 LTS*, ktorý poskytuje všetky potrebné nástroje na vývoj. Podobné kroky by však mali platiť aj na iných linuxových operačných systémoch, prípadne *MacOS* alebo *Windows 10*, ktorý v najnovších verziách podporuje *bash* a balíčkový systém *apt* [10].

Prvým krokom je nainštalovanie potrebných balíčkov príkazom *apt*.

```
sudo apt install build-essential libmpich-dev git cmake clang-format
```

Druhý krok je naklonovanie git repozitára. Tento krok môžeme preskočiť ak používame zdrojové súbory priložené k tejto práci.

```
git clone git@github.com:melias122/grid.git
```

V treťom kroku sa prepneme do repozitára a skompilujeme celý projekt. Ešte pred kompiláciou musíme vytvoriť súbor *.project-id*, ktorý musí obsahovať identifikátor pre náš projekt, v opačnom prípade nás na to upozorní príkaz *build.sh*. Prvýkrát môže byť kompilovanie časovo náročnejšie, pretože sa musí stiahnuť a skompilovať knižnica Boost s príslušnými modulmi.

```
# prepnutie sa do repozitara
cd grid/

# vytvorenie id projektu
echo 3ANTAL-2016 > .project-id

# kompilacia projektu a zavislosti
./build.sh -b
```

V štvrtom kroku by sme mali mať celý repozitár úspešne skompilovaný a môžeme založiť nový projekt. V prípade, že niečo neprebehlo dobre, môžeme všetko vrátiť do pôvodného stavu príkazom *git clean -xdf*, alebo projekt zmazať a začať od kroku jeden. Na vytvorenie nového projektu nám stačí zadať:

```
# vytvorenie noveho projektu s nazvom HelloGrid
./build.sh -n HelloGrid
```

Čo vytvorí nasledovnú štruktúru v priečinku *project*:

```
project/CMakeLists.txt
project/HelloGrid/CMakeLists.txt
project/HelloGrid/HelloGrid.pbs
project/HelloGrid/.gitignore
project/HelloGrid/src
project/HelloGrid/src/main.cpp
```

Dôležité súbory pre kompilačný systém sú všetky *CMakeLists.txt* súbory, bez ktorých sa projekt neskomkpiluje. V prípade, že by sme chceli pridať ďalší *.cpp* súbor, môžeme ho pridať do priečinku *HelloGrid/src* a následne ho musíme pridať do súboru *CMakeLists.txt*, v ktorom existuje odkaz kam pridať zdrojový súbor.

```
1 // Project HelloGrid
2 //
3 // nodes defines number of nodes to use
4 // ppn defines number of proccesors to use
5 // total cores = nodes * ppn
6 // nodes = 1
7 // ppn = 2
```

```
8  //
9  // queue defines queue for this project
10 // available queues are: serial, parallel, debug, gpu
11 // queue = parallel
12 //
13 // walltime defines time to run on grid. Format is hh:mm:ss
14 // walltime = 24:00:00
15 //
16 // Variables above are used for ../HelloGrid.pbs
17 // Edit them as needed, but do not delete them!
18 // Notice that gpus are not supported yet.
19
20 #include <iostream>
21 #include "MpiApp.h"
22 using namespace std;
23 int main(int argc, char **argv) {
24     MpiApp app(argc, argv);
25     println("Hello, " << "MpiApp");
26     return 0;
27 }
```

Výpis 15: HelloGrid/src/main.cpp

Komentáre v súbore *main.cpp* slúžia na parsovanie pre parametre *HelloGrid.pbs* súboru. Jednotlivé parametre *nodes* a *ppn* sú dôležité pre spustenie programu na lokálnom počítači. Zvyšné parametre, *queue* a *walltime* sa prejavia až pri spustení na gride. Zmeňme riadok 6 na *nodes = 2*, riadok 11 na *queue = debug*, riadok 14 na *walltime = 00:01:00* a nahraďme funkciu main nasledovne:

```
1  int main(int argc, char **argv) {
2      MpiApp app(argc, argv);
3      if (app.rank() == 0) {
4          string msg;
5          for (int i = 1; i < app.size(); i++) {
6              app.recv(i, 0, msg);
7              println(msg);
8          }
9      } else {
10         string msg("hello my number is " + to_string(Rand.Int(0, 100)));
11         app.send(0, 0, msg);
12     }
13     return 0;
14 }
```

Proces 0 v tomto príklade príjme tri správy od procesov 1, 2, 3, ktoré posielajú náhodné číslo medzi 0 a 100.

Teraz môžeme projekt *HelloGrid* skompilovať a následne spustiť pomocou príkazov:

```
# skompiluje projekt HelloGrid a vsetky zavislosti
./build.sh -b

# spustenie programu HelloGrid
./build.sh -r HelloGrid

# priklad vystupu programu HelloGrid
hello my number is 26
hello my number is 62
hello my number is 73
```

### 4.5.2 Spustenie na gride hpc.stuba.sk

Aby sme boli schopní použiť program vytvorený v kapitole 4.5.1, musíme najprv celý projekt skopírovať na grid hpc.stuba.sk. Možností ako skopírovať projekt je viacero. Môžeme ho skopírovať manuálne, napríklad pomocou programu *rsync*, *scp* alebo môžeme použiť systém *git*, prípadne iný program na kopírovanie. Pri manuálnom kopírovaní si treba dať pozor na to, aby sme neskopírovali priečinky a súbory, ktoré závisia od lokálneho prostredia. Sú to hlavne súbory z predchádzajúcich kompilácií a priečinky: *.build* a *vendor/boost*. Odporúčame preto použiť príkaz *build.sh* a jeho argument *--sync*. Nachádzame sa v priečinku *$HOME/grid* na lokálnom počítači a repozitár zosynchronizujeme pomocou:

```
./build.sh --sync
```

Príkaz *./build.sh --sync* skopíruje cely projekt na server hpc.stuba.sk do priečinku */work/3xelias/grid*. Taktiež skopíruje *.project-id* a vynechá všetky súbory viazané na lokálne prostredie.

Po skopírovaní súborov sa môžeme prihlásiť pomocou *ssh*.

```
# prihlasenie do gridu
ssh 3xelias@login.hpc.stuba.sk

# prepnutie sa do priecinku kam sme skopirovali projekt
cd /work/3xelias/grid

# skompilovanie projektu
./build.sh -b
```

```
# pridanie programu do vypoctovej fronty
./build.sh -r project/HelloGrid
```

V tomto prípade neuvidíme žiadny výstup. Výstup si môžeme pozrieť až po zaradení programu do fronty a po jeho skončení, čo by malo v prípade, že sme použili *debug* frontu takmer okamžité. Výstup môžeme nájsť po skončení programu v priečinku *project/HelloGrid*. Sú to súbory *HelloGrid* s príponou *.o<id_procesu>* pre štandardný výstup a *.e<id_procesu>* pre chybový výstup.

Výstupy si môžeme stiahnuť na lokálny počítač pomocou:

```
# exit z gridu
exit

# na lokalnom pocitaci v priecinku $HOME/grid
./build.sh --sync
```

Tentokrát, ak sme nespravili žiadnu zmenu na lokálnom počítači, tak sa neskopíruje nič, ale stiahnu sa výstupy z programu *HelloGrid* do priečinku *project/HelloGrid*.

# 5 Experiment s GA

V našej práci sme sa rozhodli vyskúšať experiment s genetickými algoritmami, ktoré lúštia monoalfabetickú substitúciu. Dôležitosť tohto experimentu spočívala v nájdení vhodných parametrov, pri ktorých bude najvyššia úspešnosť rozlúštenia zašifrovaného textu.

Prvým krokom bolo nasadenie vlastnej implementácie GA, ktoré by sme mohli upraviť podľa vlastných požiadaviek.

Ďalším krokom bolo zvolenie vhodných parametrov GA, pre nájdenie optimálneho riešenia. Naším zámerom bolo preskúmať viacero parametrov, ktoré by sme mohli neskôr použiť pre ďalšie skúmanie paralelných genetických algoritmov. Medzi skúmané parametre patrili: veľkosť počiatočnej populácie, počet iterácií genetického algoritmu a kombinácie rôznych operácií (schémy). Veľkosť počiatočnej populácie, ktorú sme skúmali bola v počte 10, 20, 50 a 100 chromozómov. Počet iterácii bol 10000 a 50000. Zvolené schémy môžeme vidieť v tabuľke 5.

Prvým stĺpcom tabuľky je názov samotnej schémy. Ďalšie stĺpce tabuľky predstavujú zreťazenie jednotlivých operácií. Riadky schémy predstavujú aplikovanie jednotlivých reťazcov operácií na populáciu z predchádzajúcej generácie. Všimnime si napríklad schému *J*. Prvou operáciou v prvom riadku tejto schémy je výber elitného jedinca, ktorý postupuje do novej generácie. V druhom riadku je tak isto zvolený jeden elitný jedinec, ale tentokrát je nad ním vykonaná operácia mutácie, ktorá jeden krát zamení pozície dvoch náhodne vybraných génov. V treťom riadku schémy *J* sa *((n-2)/2)*-krát vykoná náhodný výber, ktorého výstupom sú dvaja jedinci. Títo jedinci postupujú do operácie kríženia, ktorej výsledkom sú dvaja noví potomkovia, nad ktorými sa vykoná operácia swap.

Úspešnosť jednotlivých schém bola vyhodnocovaná nad textami od dĺžky 50 po 2000 s nárastom o 50 znakov, pričom z každej dĺžky sme mali k dispozícii 100 rôznych textov. Po celom behu GA bol vybratý najlepší jedinec reprezentujúci potencionálny kľúč, ktorým bol dešifrovaný zašifrovaný text, a jeho výsledok porovnaný s pôvodným otvoreným textom. Ako ohodnocovacia funkcia bola použitá bigramová funkcia, ktorá vypočítavala manhatanovskú vzdialenosť. Táto funkcia bola zvolená ako východisková, pretože skúmanie rôznych fitness funkcií je veľmi rozsiahle a je nad rámec tejto práce.

## 5.1 Distribúcia parametrov

Aby sme mohli využiť celú výpočtovú silu ktorú nám gridové prostredie ponúka a aby sme sa zároveň vyhli opakovanému spúšťaniu každej schémy, navrhli sme riešenie, v

| Schéma | Výber | Mutácia | Kríženie | Suboperácia |
|---|---|---|---|---|
| **A** | Tournament(n) | Swap(1) | - | - |
| **B** | Random(n) | Swap(1) | - | - |
| **C** | Tournament(n/2) | Swap(1) | - | - |
| | Tournament(n/2) | - | - | - |
| **D** | Tournament(n/2) | Swap(1) | - | - |
| | Random(n/2) | - | - | - |
| **E** | Elite(1) | - | - | - |
| | Elite(1) | Swap(1) | - | - |
| | (n-2) * Tournament(2) | - | Singlepoint(2) | |
| **F** | Elite(1) | - | - | - |
| | Elite(1) | Swap(1) | - | - |
| | (n-2) * Tournament(2) | - | Singlepoint(1) | - |
| **G** | Elite(1) | - | - | - |
| | Elite(1) | Swap(1) | - | - |
| | ((n-2)/2) * Tournament(2) | - | Singlepoint(2) | Swap(1) |
| **H** | Elite(1) | - | - | - |
| | Elite(1) | Swap(1) | - | - |
| | (n-2)*Random(2) | - | Singlepoint(1) | - |
| **I** | Elite(1) | - | - | - |
| | Elite(1) | Swap(1) | - | - |
| | (n-2)/2 * Random(2) | - | Singlepoint(2) | - |
| **J** | Elite(1) | - | - | - |
| | Elite(1) | Swap(1) | - | - |
| | ((n-2)/2) * Random(2) | - | Singlepoint(2) | Swap(1) |

Tabuľka 5: Schémy GA

ktorom by sme mali jeden hlavný proces distribuujúci parametre n pracovným procesom. Tento spôsob môže mať dve riešenia. Prvým je, že hlavný proces rozhoduje komu posiela prácu. Druhý spôsob riešenia je, že pracovné procesy notifikujú hlavný proces, keď sú voľné. Nevýhodou prvého riešenia je, že môžu vznikať časové prestoje pracovných procesov, pretože tu chýba spätná komunikácia a hlavný proces prakticky nemá možnosť zistiť, či proces svoju prácu skončil. Tento problém rieši druhý prístup, kde sa pracovné procesy snažia získať prácu hneď, ako je to možné. Preto sme sa rozhodli implementovať druhý spôsob, ktorý je podľa nášho názoru ľahší na implementáciu a aj rýchlejší.

Úlohou hlavného procesu je distribúcia parametrov (nastavení GA) pracovným procesom. Hlavný proces zostaví parametre: iterácia, veľkosť populácie, dĺžka textu, načíta otvorený text a zašifrovaný text. Potom čaká na notifikáciu od ľubovoľného pracovného procesu, od ktorého príjme jeho id. Následne zostaví parametre a pošle ich pracovnému procesu na spracovanie. V prípade, že hlavný proces už nemá čo poslať, čaká na všetky pracovné procesy, ktoré následne ukončí.

Úlohou pracovného procesu je na prijaté parametre aplikovať schémy z tabuľky 5. Pracovný proces sa snaží získať prácu od hlavného procesu tak, že mu najprv pošle svoje id a potom čaká kým mu hlavný proces pridelí prácu (parametre), alebo zruší jeho činnosť.

```
1  void run_master(mpi::comm &comm)
2  {
3      for (int itteration : {10000, 50000} ) {
4          for (int populationSize : {10, 20, 50 , 100}) {
5              for (int textsize = 50; textsize <= 2000; textsize += 50) {
6                  for (int text = 1; text <= 100; text++) {
7                      string pt = read_plaintext();
8                      string ct = read_ciphertext();
9                      Data data;
10                     comm.recv(mpi::any_source, 0, data);
11                     // copy data
12                     data.itteration = itteration;
13                     ...
14                     comm.send(data.workerId, 1, data);
15                 }
16             }
17         }
18     }
19     // shutdown workers
20 }
21
22 void run_worker(int id, mpi::comm &comm)
```

```
23 {
24     // initialize
25     while (1) {
26         Data data(id);
27         // nonblocking send
28         comm.isend(master, 0, data);
29         comm.recv(master, 1, data);
30         if (!data.status) return;
31         // setup GA scheme
32         Scheme scheme = ...
33         for (schemeId &s : {A, B, C, ...}) {
34             GeneticAlgorithm::run(scheme);
35             // write best
36         }
37     }
38 }
```

Výpis 16: Pseudokód ditribúcie parametrov GA

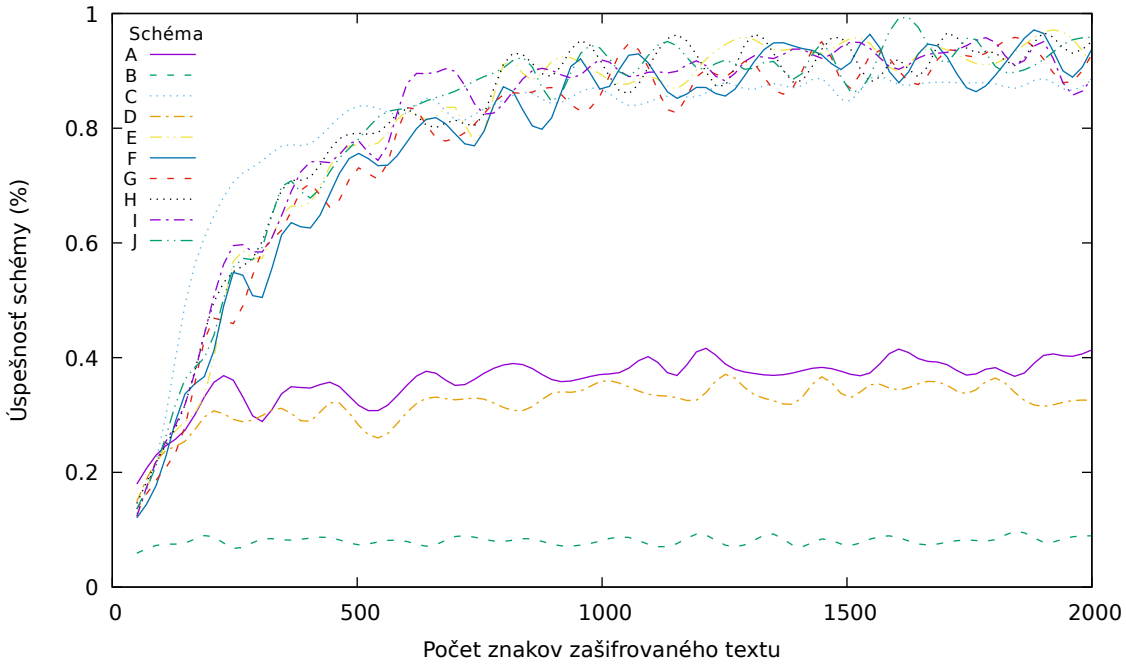
## 5.2 Výsledky experimentu

Experiment s GA bežal na gride hpc.stuba.sk 34 hodín. Použitých bolo pritom 96 výpočtových uzlov, čo zjavne urýchlilo celý experiment. Spolu bolo treba 320000 spustení genetického algoritmu.

Na obrázku 3 môžeme vidieť výsledky genetického algoritmu pre najmenšie skúmané nastavenia 10000 iterácií a 10 jedincov. Ako možno vidieť z grafu, najhoršie výsledky dosiahla schéma B, ktorá je zložená z náhodného výberu a jedného swapu. Táto schéma sa správa podľa našich očakávaní. Náhodnosť spôsobuje vysokú mieru diverzity. O niečo lepšie, ale podobné výsledky dosahujú schémy A a D, ktoré sa správajú veľmi náhodne a neprinášajú lepšie výsledky. Prekvapivé výsledky prináša schéma C, v ktorej vidieť vysokú mieru selektívneho tlaku, avšak táto schéma, ako vidieť z grafu, uviazne v nejakom lokálnom extréme. Lepšie výsledky prinášajú schémy E, F, G, H, I J, ktoré vďaka zachovaniu elitných jedincov dokážu pri dostatočne dlhých textoch prinášať dobré výsledky.

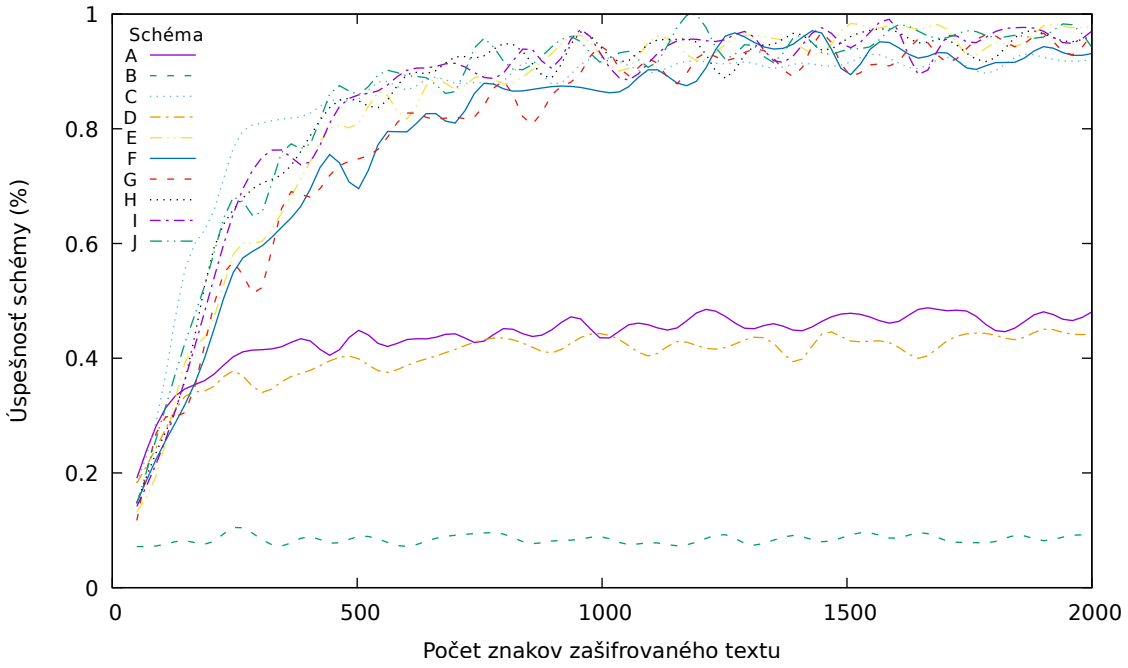
Na ďalšom obrázku 4 môžeme vidieť GA s počiatočnou populáciou veľkosti 50 jedincov. Zvýšenie počtu jedincov spôsobuje malé zlepšenie „náhodných“ schém A a D, avšak úspešnosť týchto schém nie je dostatočná. Zvýšenie veľkosti populácie značne pomohlo schéme C, ktorá takto vďaka vyššej diverzite má možnosť dosiahnuť globálny extrém. Zvyšné schémy, ktoré obsahujú elitných jedincov, dosahujú s vyšším počtom jedincov mierne zlepšenie úspešnosti v kratších textoch.


Obrázok 3: Závislosť úspešnosti lúštenia od dĺžky ZT (10000 iterácií, 10 jedincov)




Obrázok 4: Závislosť úspešnosti lúštenia od dĺžky ZT (10000 iterácií, 50 jedincov)

Podľa našich očakávaní boli dosiahnuté najlepšie výsledky s nastavením genetického algoritmu pri 50000 iterácií a s počiatočnou populáciou veľkosti 100 jedincov (obrázok 5). Lepšie výsledky boli dosiahnuté najmä pri kratších textoch a mierne zlepšenie môžeme vidieť aj pri dlhších textoch. Treba však poznamenať, že GA pri taktom nastavení beží značne dlhšie a dobré výsledky sa dajú dosiahnuť aj s menším počtom iterácií a menšou populáciou.



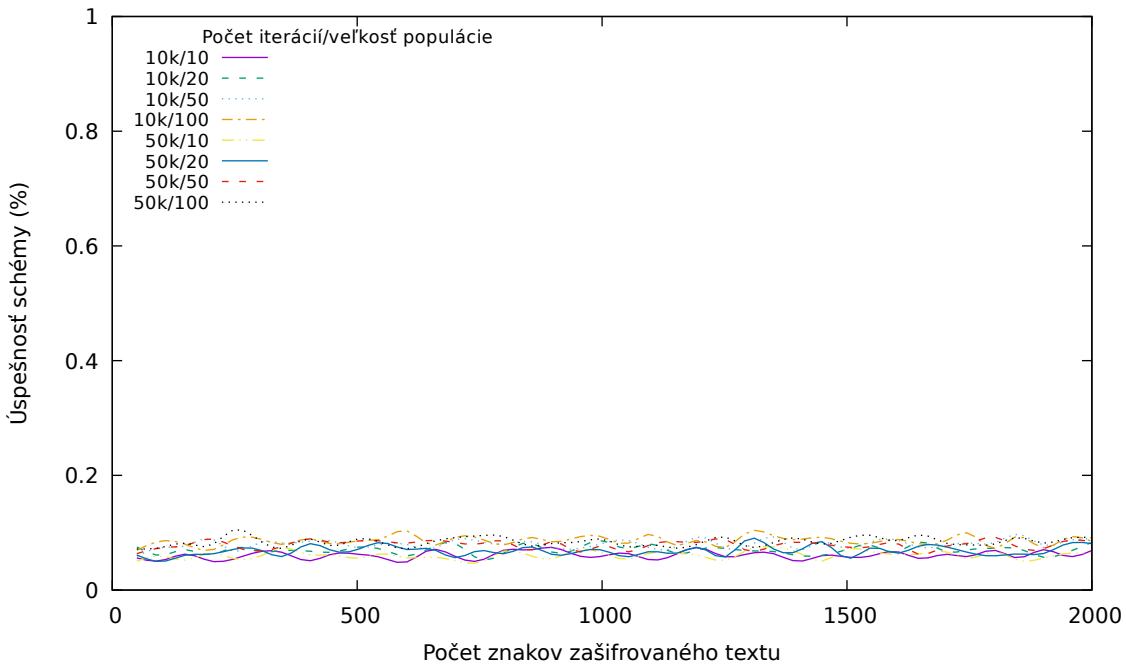

Obrázok 5: Závislosť úspešnosti lúštenia od dĺžky ZT (50000 iterácií, 100 jedincov)

Zo schémy B na obrázku 6 možno vidieť, že bez absencie selektívneho tlaku GA nekonverguje k lepším výsledkom.

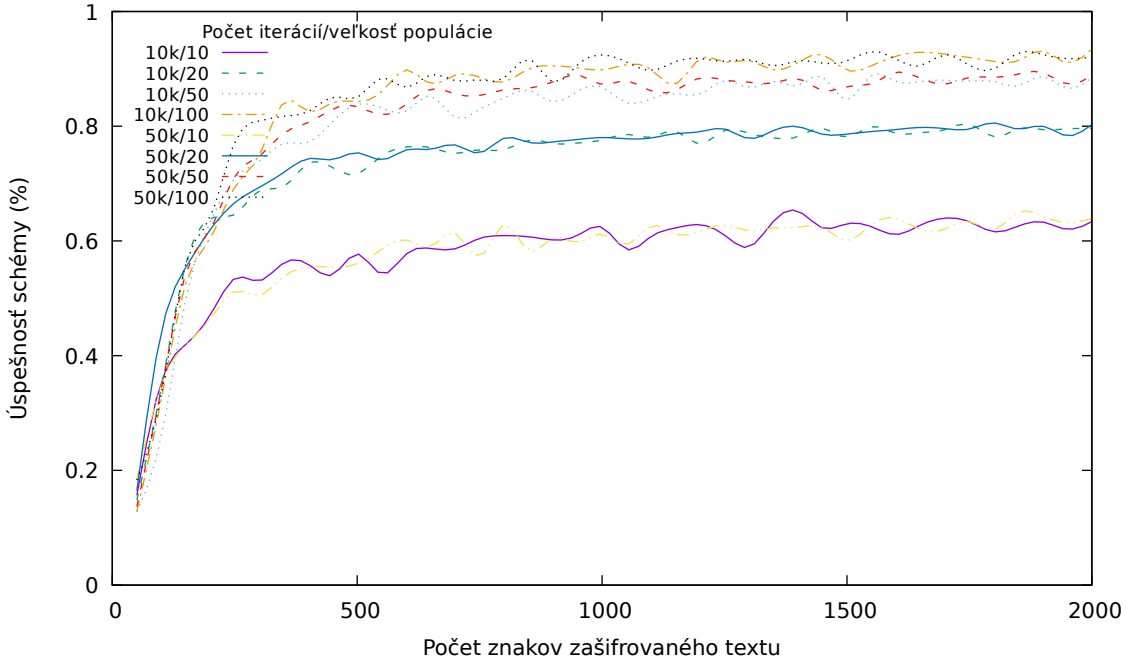
Ako sme už poznamenali, schéma C (obrázok 7) dosahuje prekvapivé výsledky aj napriek tomu, že neobsahuje výber elitných jedincov. K dosahovaniu vysokej úspešnosti jej pomáha najmä veľkosť populácie, avšak ako možno vidieť z grafu tejto schémy, napríklad zväčšenie populácie z 50 a 100 jedincov už neprinieslo o toľko lepšie výsledky.

Na ďalších obrázkoch schém E (obrázok 8) a J (obrázok 9) môžeme vidieť, že zväčšenie počtu iterácií GA ako aj veľkosti populácie prináša pri použití týchto schém iba mierne zlepšenie úspešnosti. Z grafov týchto funkcií vyplýva, že zlepšenie sa pohybuje okolo 10%.

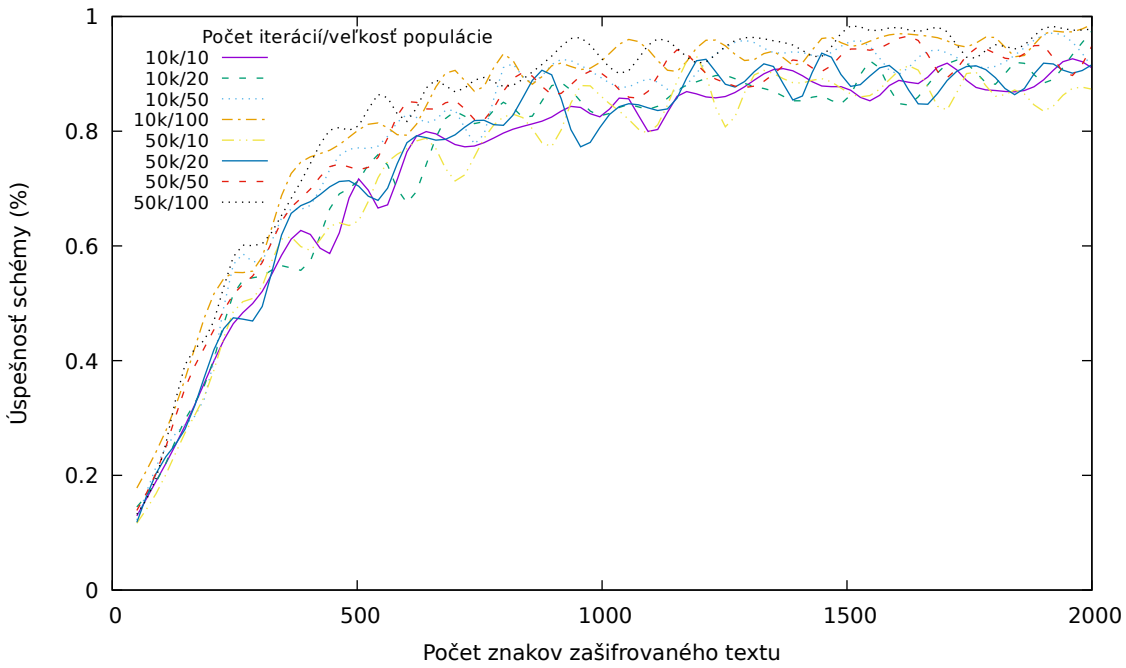
Z našich výsledkov vyplýva, že schémy C, E a J dosahujú najlepší výsledok pri lúštení monoalfabetickej substitúcie, okrem toho odporúčame aspoň 20 jedincov a aspoň 10000 iterácií GA. Zvyšné výsledky tohto experimentu sa nachádzajú v prílohe tejto práce.




Obrázok 6: Závislosť úspešnosti lúštenia od dĺžky ZT (schéma B)




Obrázok 7: Závislosť úspešnosti lúštenia od dĺžky ZT (schéma C)



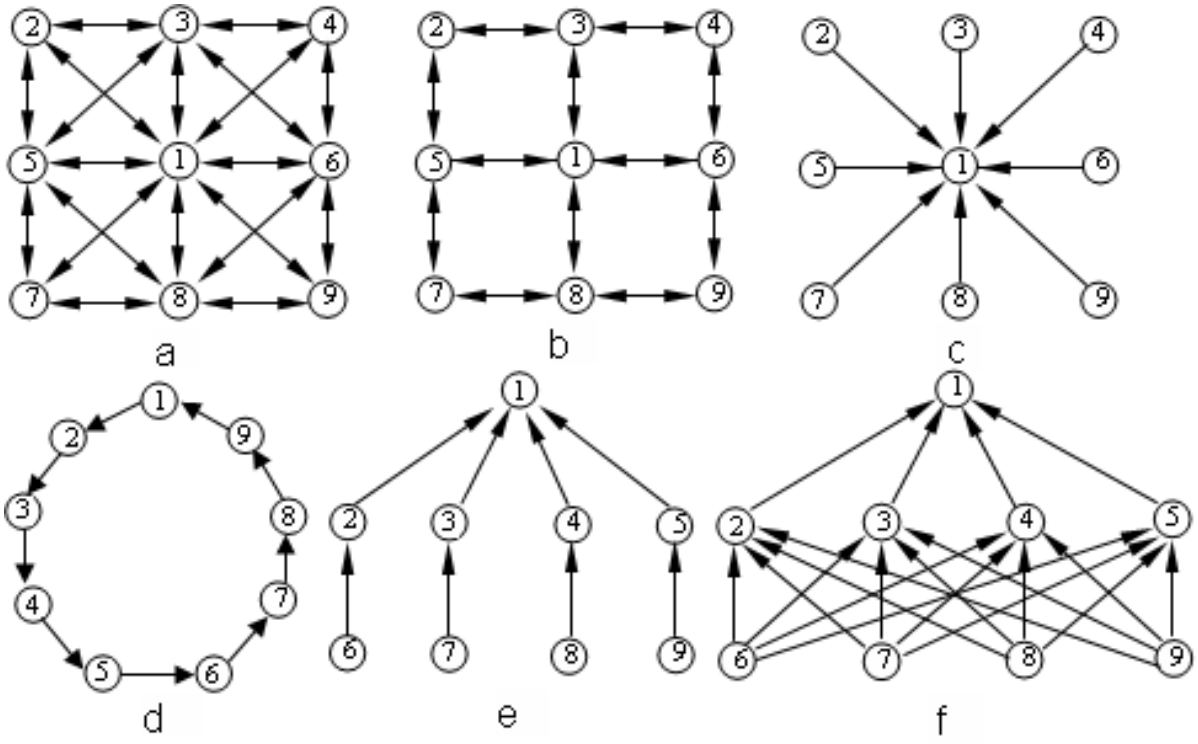

Obrázok 8: Závislosť úspešnosti lúštenia od dĺžky ZT (schéma E)


Obrázok 9: Závislosť úspešnosti lúštenia od dĺžky ZT (schéma J)

# 6 Experiment s PGA

Ďalší experiment, ktorý sme vykonali, bol experiment s paralelnými genetickými algoritmami. V tomto experimente sme sa zamerali na skúmanie rôznych topológií s rôznym tvarom, počtom ostrovov a migračných časov. Ako schémy sme použili C, E a F z prechádzajúceho experimentu. Z obrázku 10 boli zvolené topológie b, d, e, f. Počet ostrovov bol zvolený 3, 5 a 11. Pre veľkosti 3 a 11 bola topológia b mierne upravená. V prípade 3 ostrovov mala schéma b trojuholníkový tvar a v prípade 11 ostrovov bol k 4 a 6 uzlu pridaný ďalší štvorec. Počet iterácií GA bol zvolený 30000, pričom jednotlivé ostrovy migrovali 30, 6 a 3 krát, teda každých 1000, 5000 a 10000 iterácií.

Vyhodnocovanie PGA prebiehalo podobne ako v predchádzajúcom experimente. Vyhodnocovali sme texty dĺžky od 50 do 2000 s nárastom o 50 znakov s rozdielom že teraz sme skúmali len 75 textov. Vyhodnocovaný bol najlepší jedinec zo všetkých populácií v prípade topológie b, d. Pri topológiách e, f bol vyhodnotený vrchol stromu.



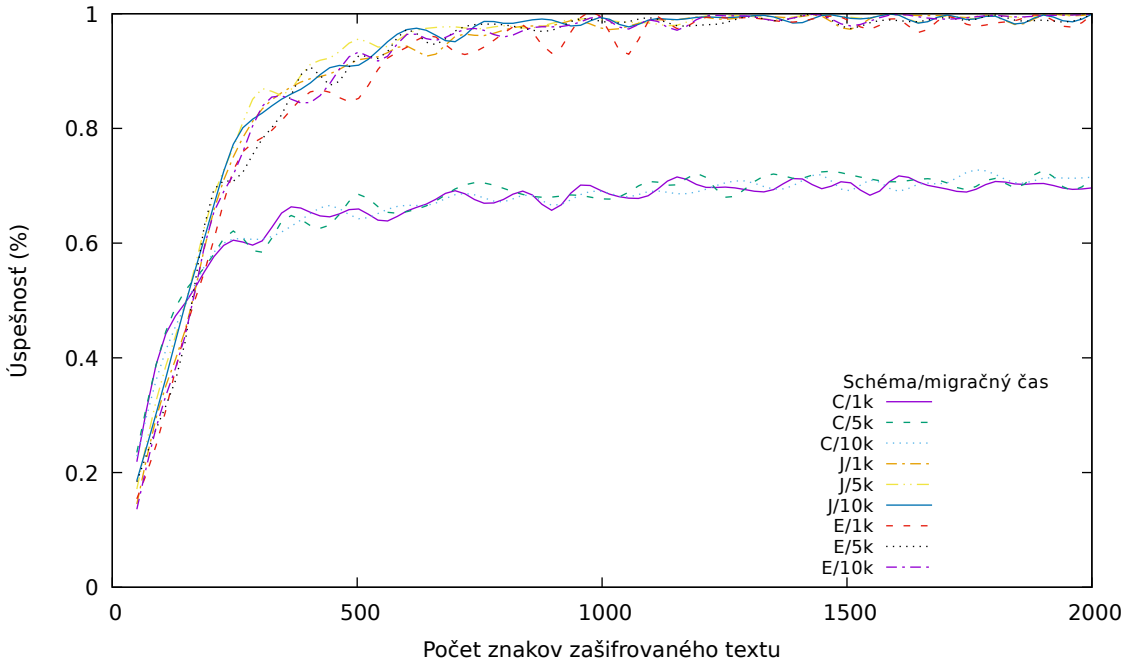

Obrázok 10: Topológie paralelných genetických algoritmov [1]

## 6.1 Výsledky experimentu

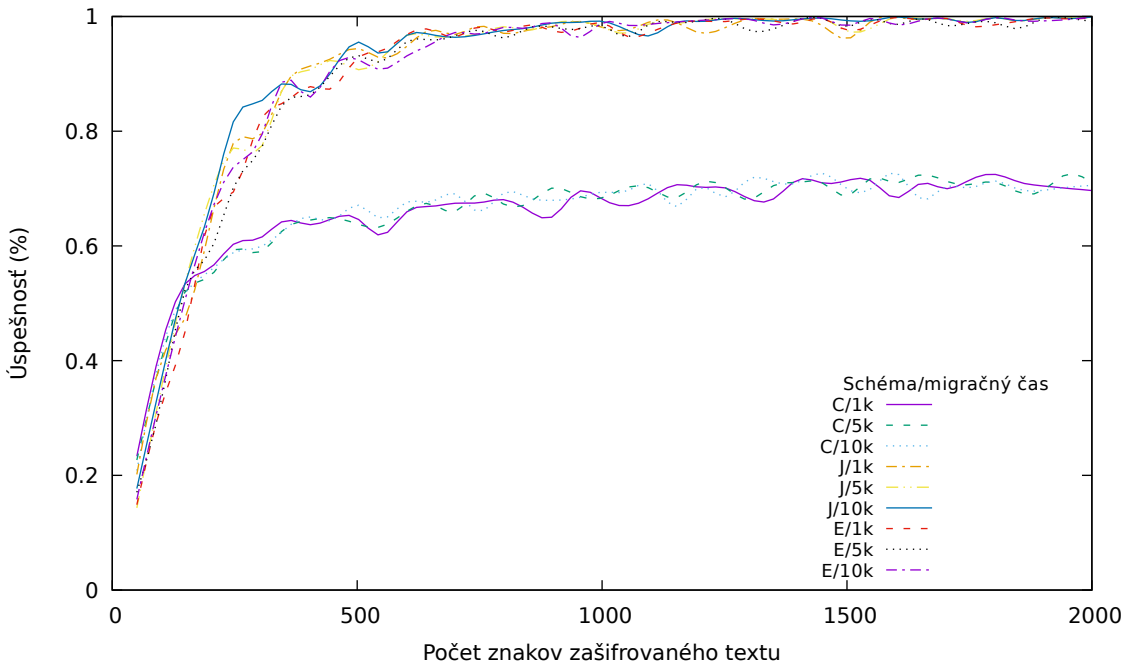
Experiment s PGA bežal približne dva aj pol týždňa. K vykonaniu tohto experimentu bolo treba 7884000 spustení jednotlivých GA.

Na obrázkoch 11 a 12 môžeme vidieť výsledky z experimentu pre topológie *b* a *d*. Z týchto výsledkov vyplýva, že v nami riešenom experimente nezáleží na topológii PGA. Avšak oproti GA výsledky dosahujú lepšiu úspešnosť. V prípade schém *E* a *J* dosahujeme 80% úspešnosť už pri textoch dĺžky 250 znakov a takmer 100% úspešnosť pre texty dlhšie ako 1000 znakov. Schéma *C* dosahuje asi o 10% lepšiu úspešnosť pre PGA. Ostatné výsledky sú veľmi podobné a sú priložené ako príloha k tejto práci.

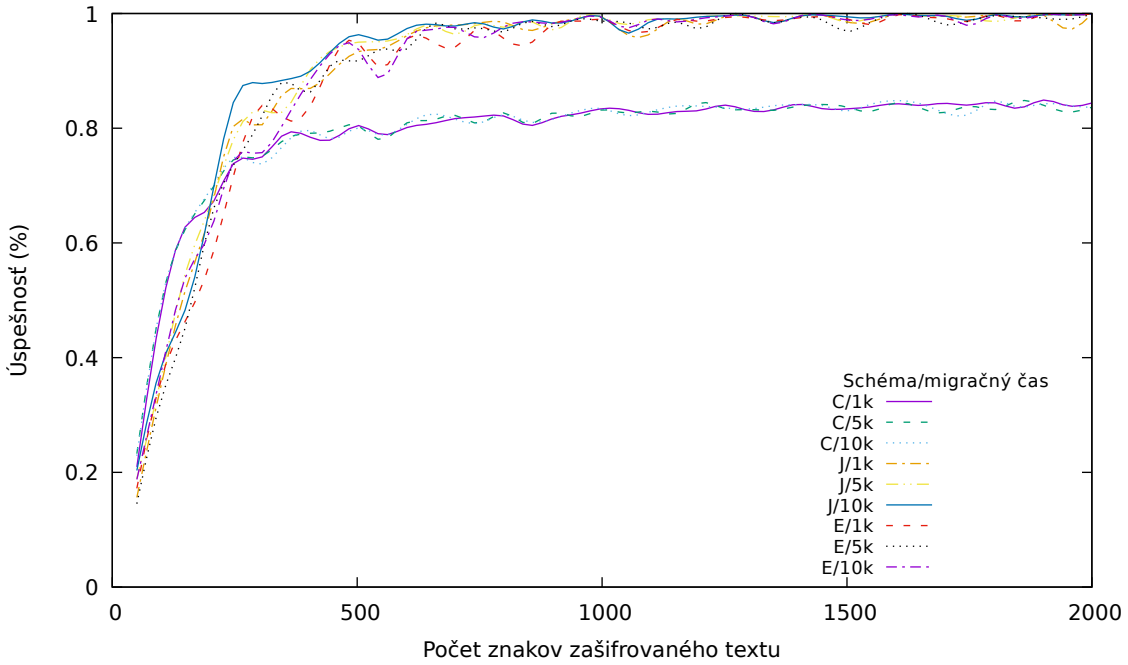
Z výsledkov na obrázkoch 13, 14 a 15 vidieť, že veľkosť populácie pozitívne vplýva na výsledky PGA. Výsledky sú mierne zlepšené veľkosťou topológie. Veľký rozdiel v úspešnosti môžeme vidieť medzi populáciou 20 (obrázok 13) a 50 (obrázok 14) najmä v schéme *C*, ale aj v prípade schém *E* a J.



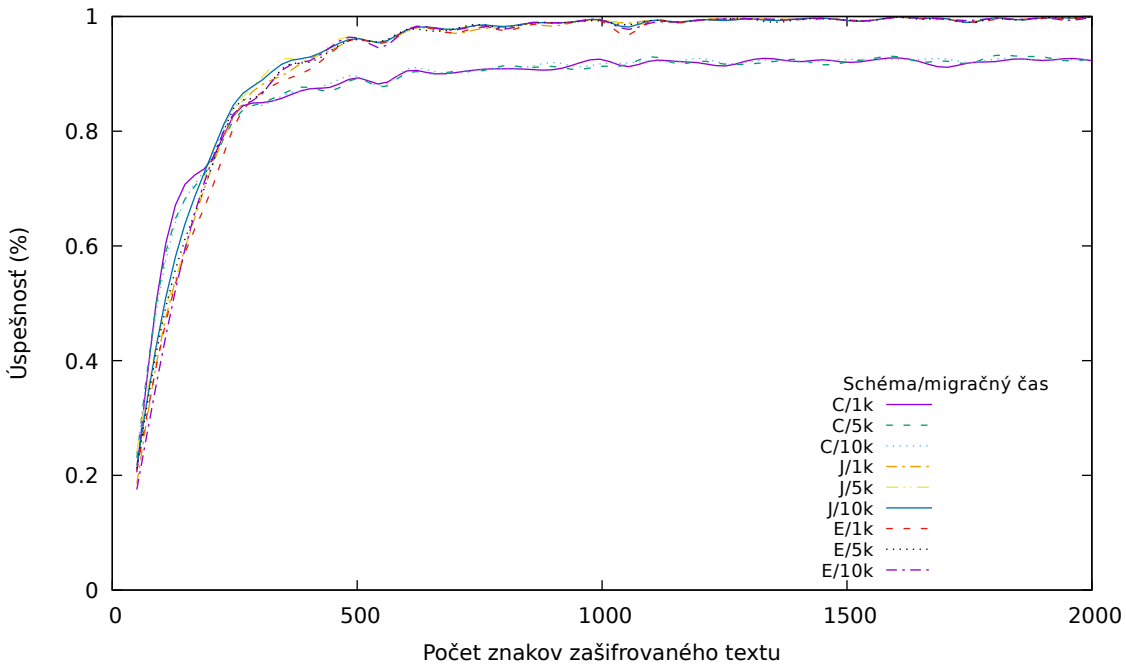

Obrázok 11: Závislosť úspešnosti lúštenia od dĺžky ZT (topológia b/3, 10 jedincov)



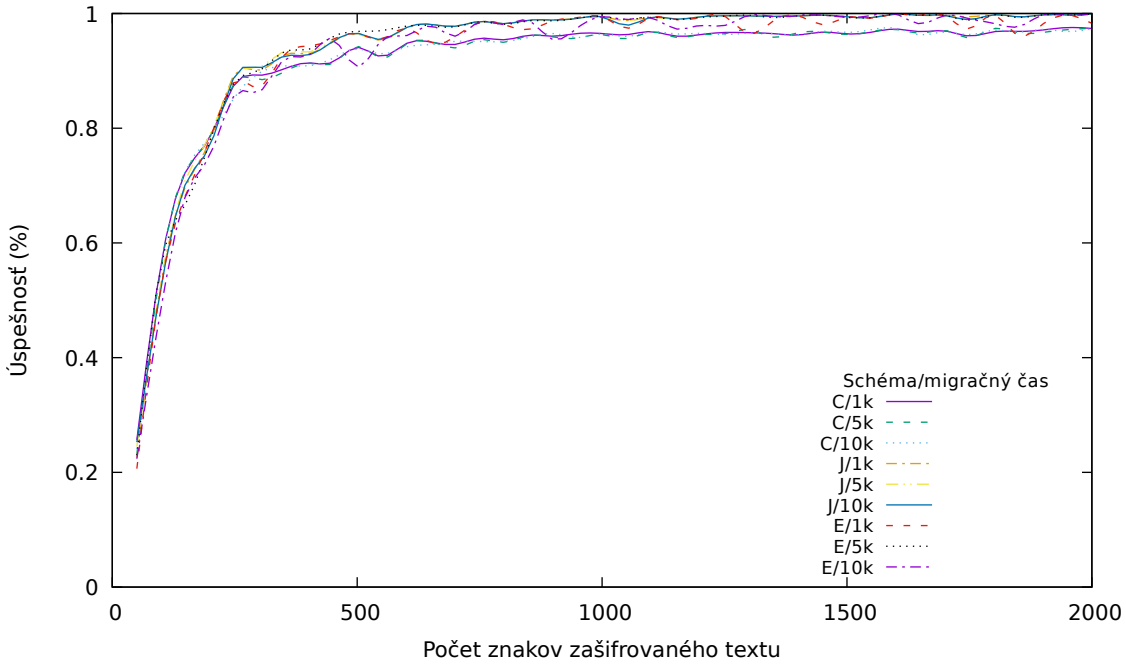

Obrázok 12: Závislosť úspešnosti lúštenia od dĺžky ZT (topológia d/3, 10 jedincov)




Obrázok 13: Závislosť úspešnosti lúštenia od dĺžky ZT (topológia b/3, 20 jedincov)




Obrázok 14: Závislosť úspešnosti lúštenia od dĺžky ZT (topológia b/5, 50 jedincov)




Obrázok 15: Závislosť úspešnosti lúštenia od dĺžky ZT (topológia b/11, 100 jedincov)

# Záver

Cieľom diplomovej práce bolo vytvorenie nástrojov na kryptoanalýzu klasických šifier v gridovom prostredí SIVVP. Tento krok bol úspešne zrealizovaný. Vytvorili sme vývojové prostredie, ktoré zjednodušuje prácu a vývoj v lokálnom prostredí a prostredníctvom ktorého je možné: vytvárať projekty, synchronizovať ich s gridom a spúšťať lokálne a tiež na gride. V nami vyvinutom vývojovom prostredí sme vytvorili modul pre paralelné genetické algoritmy, ktorý si ďalší lúštitelia môžu prispôsobiť podľa vlastných potrieb.

Tento modul sme si aj my upravili podľa vlastných potrieb a implementovali sme útok na monoalfabetickú substitúciu pomocou PGA. V prvom experimente sme hľadali vhodné schémy a nastavenia GA, pričom sme využili distribúciu parametrov na 96 uzloch gridu. Tento experiment trval 34 hodín. Výsledkom experimentu bolo, že sedem z desiatich schém dosahovalo 80% úspešnosť pri textoch dĺžky 500 znakov a viac ako 90% úspešnosť pri textoch dlhších ako 1000 znakov.

Schémy z prvého experimentu sme použili v nasledujúcom experimente s PGA. Na týchto schémach sme skúmali vplyv štyroch rôznych topológií a troch migračných časov, a ich úspešnosť pri lúštení monoalfabetickej substitúcie. Expriment trval približne dva aj pol týždna. Podľa experimentov sme zisli, že rôzne topológie nemajú takmer žiadny vplyv na zvýšenie úspešnosti riešenia monoalfabetickej substitúcie. Na úspešnosť riešenia vplýva najmä veľkosť populácie a mierne aj veľkosť topológie. Úspešnosť riešenia pomocou PGA dosahovala pre texty dĺzky 500 a viac znakov takmer 100%, ale lepšie výsledky boli dosiahnuté aj pri kratších textoch.

Počas vytvárania tejto práce sme spolupracovali aj s Bc. Petrom Javorkom, ktorému sme ingtegrovali jeho riešenie na grid pomocou nami vytvoreného softvéru. Na experimenty sme spotrebovali takmer 70000 procesorových hodín, čo by nás stálo približne 700 eur.

# Zoznam použitej literatúry


1. SEKAJ, Ivan a ORAVEC, Michal. Paralelné evolučné algoritmy. *Umelá inteligencia a kognitívna veda III.* 2011, s. 243–267.
2. SEKAJ, Ivan. *Evolučné výpočty a ich využitie v praxi.* Iris, 2005. ISBN 9788089018871.
3. GROŠEK, Otokar, VOJVODA, Milan a ZAJAC, Pavol. *Klasické šifry.* Slovenská technická univerzita, 2007. ISBN 978-80-227-2653-5.
4. KERCKHOFFS, A. a COLLECTION, George Fabyan. *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef.* Librairie militaire de L. Baudoin, 1883. Extrait du Journal des sciences militaires.
5. *STUBA klaster - IBM iDataPlex* Dostupné na internete: <`https://www.hpc.stuba.sk`>.
6. *Commands overview.* Adaptive Computing, 2012 Dostupné na internete: <`http://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/12-appendices/commandsOverview.htm`>.
7. SNIR, Marc, OTTO, Steve, HUSS-LEDERMAN, Steven, WALKER, David a DONGARRA, Jack. *MPI-The Complete Reference, Volume 1: The MPI Core.* 2nd. (Revised). Cambridge, MA, USA: MIT Press, 1998. ISBN 0262692155.
8. MESSAGE PASSING INTERFACE FORUM. *MPI: A Message-Passing Interface Standard.* 2015 Dostupné na internete: <`http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`>.
9. *CMake: Reference Documentation.* 2017 Dostupné na internete: <`https://cmake.org`>.
10. *Bash on ubuntu on Windows.* 2017 Dostupné na internete: <`https://msdn.microsoft.com/commandline/wsl/about`>.

# Prílohy

A Štruktúra elektronického nosiča . . . . . II

# A Štruktúra elektronického nosiča

*/diplomovka.pdf*

· Tento dokument

*/priloha.zip*

· Príloha k dokumentu