

**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

Martin Eliáš

Podakovanie

I would like to express a gratitude to my thesis supervisor.

Obsah

Úvod	1
1 Klasické šifry	2
1.1 História	2
1.2 Charakteristika	4
1.3 Počítačové lúštenie klasických šifier	5
1.3.1 Útok hrubou silou	6
1.3.2 Slovníkový útok	7
1.3.3 Horolezecký algoritmus	7
1.3.4 Genetické algoritmy	8
2 Grid	11
2.1 hpc.stuba.sk	11
2.2 Príkazy	12
2.2.1 module	13
2.2.2 qstat	13
2.2.3 qfree	14
2.3 Výpočtové fronty	15
2.4 Príklad sériovej úlohy	16
2.5 Príklad paralelnej úlohy	17
3 MPI	20
3.1 Point-to-point komunikácia	20
3.2 Blokujúca komunikácia	22
3.3 Neblokujúca komunikácia	23
3.4 Dynamická alokácia	25
3.5 Serializácia dátových typov	26
3.6 Boost MPI	26
4 Návrh	27
4.1 Štruktúra prostredia	27

4.2	Implementácia	28
4.3	Príklad použitia	30
4.3.1	Lokálny vývoj	30
4.3.2	Spustenie na gride hpc.stuba.sk	32
5	Experiment s GA	34
5.1	Distribúcia parametrov	36
5.2	Výsledky	38
6	Experiment s PGA	43
6.1	MpiMigrator	44
6.2	Topológie	45
6.3	Výsledky experimentu	45
	Záver	47
	Zoznam použitej literatúry	48
	Prílohy	I
	A Štruktúra elektronického nosiča	II
	B Výsledky Genetického algoritmu	III

Zoznam obrázkov a tabuliek

Obrázok 1	Štruktúra GA	9
Obrázok 2	Počet iterácii: 10000, počiatočná populácia: 10	38
Obrázok 3	Počet iterácii: 10000, počiatočná populácia: 50	39
Obrázok 4	Počet iterácii: 50000, počiatočná populácia: 100	40
Obrázok 5	Schéma: B	40
Obrázok 6	Schéma: C	41
Obrázok 7	Schéma: E	42
Obrázok 8	Schéma: J	42
Obrázok 9	Štruktúra PGA s migračným modelom	43
Obrázok 10	Topológie paralelných genetických algoritmov	46
Obrázok B.1	Počet iterácii: 10000, počiatočná populácia: 20	III
Obrázok B.2	Počet iterácii: 10000, počiatočná populácia: 100	IV
Obrázok B.3	Počet iterácii: 50000, počiatočná populácia: 10	IV
Obrázok B.4	Počet iterácii: 50000, počiatočná populácia: 20	V
Obrázok B.5	Počet iterácii: 50000, počiatočná populácia: 50	V
Obrázok B.6	Schéma: A	VI
Obrázok B.7	Schéma: D	VI
Obrázok B.8	Schéma: F	VII
Obrázok B.9	Schéma: G	VII
Obrázok B.10	Schéma: H	VIII
Obrázok B.11	Schéma: I	VIII
Tabuľka 1	Disky	12
Tabuľka 2	Výpočtové fronty a ich obmedzenia	16
Tabuľka 3	Dátové typy v MPI a ich C ekvivalenty	24
Tabuľka 4	schemy	35

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 Interfac
PBS	Portable Batch System
RAM	Random Access Memory
SSH	Secure shell
VPN	Virtual private network

Zoznam výpisov

1	module avail	13
2	qstat	13
3	qstat -u 3xelias	14
4	qfree	14
5	uloha1.pbs	16
6	uloha2.pbs	17
7	Point-to-point komunikácia	21
8	MPI_Send	22
9	MPI_Recv	23
10	Neblokujúca komunikácia	24
11	Štruktúra prostredia	27
12	HelloGrid/src/main.cpp	31
13	Pseudokód distribúcie parametrov GA	36
14	Pseudokód MpiMigrator	45

Úvod

Tu bude krásny úvod s diakritikou atď.

A možno aj viac riadkový úvod.

1 Klasické šifry

V tejto kapitole sa budeme zaoberať históriou a stručným prehľadom klasických šifier. Spomenieme si aj niektoré základné útoky na klasické šifry.

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 Aremejci a po nich následné Gréci, aby pomocou nich boli schopný rozlíšiť jazyky [1]. Aj diakritika ako taká má schopnosť rozlišovať významy slov, čo si ale až do 15. storočia nikto nevšimol, 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 vynášiel 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 široké rozšírenie abecedy a písomného prejavu. Gréci písmo prebrali od Feničanov, ktorí na rozdiel od Egyptanov 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 samoh-

lások [1].

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 použitý 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 sústreďenie

Tieto princípy sú popísané v pôvodnej publikácii od Kerckhoffa [2].

Existovala ale aj iná skupina vedcov, medzi ktorých patrila aj Lester S. Hill, ktorý si uvedomoval, že kryptológia je úzko spätá z matematikou. V roku 1918 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].

1.2 Charakteristika

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

- 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.

Zo spomenutých charakteristík existujú aj výnimky. Napríklad pri Vigenereovej 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 [1].

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 ot-

voreného/zašifrovaného textu. Od polyalfabetickej šifry sa líši v poradí vykonávania operácií.

- **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 substitučný 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, ktorá sa aplikuje na celý blok. 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 čias-točnej, alebo až úplnej kryptoanalýzy klasických šifier“ [1]. Takúto automatizovanú kryptoanalýzu môžeme rozdeliť do troch kategórií:

1. Prvou kategóriou je počítačom asistovaná kryptoanalýza, pri ktorej lúštitel využíva počítač ako pomôcku na zjednodušenie niektorých úkonov.
2. Druhou kategóriou je poloautomatická kryptoanalýza, pri ktorej počítač prehľadáva priestor riešení, avšak rozhodujúce kroky sú prenechané na kryptoanalytika.
3. Poslednou treťou kategóriou je automatická kryptoanalýza, pri ktorej lúštitel zadá zašifrovanú správu počítaču a ten mu následne poskytne jednu, prípadne aj viac najlepších riešení.

Automatickou kryptoanalýzou sa budeme zaoberať aj v tejto práci. Pri automatickej kryptoanalýze zohráva veľkú úlohu analýza a ohodnocovanie textov, pre-

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

tože 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 metódy, 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 extrémny. 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: horoložecký algoritmus, simulované žihanie, 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 „VECDOXSORS CDYBSMUIMRCSPSOBXKQBSNO“, ktorý vieme že bol zašifrovaný šifrou podobnou Cézarovej šifry. Pre získanie otvoreného textu potrebujeme vyskúšať všetkých 26 možností posunov, čo je v tomto prípade kľúč, tak, aby sme dostali zmysluplný text.

kluc 1

```
VECDOXSORS CDYBSMUIMRCSPSOBXKQBSNO  
WFDEPYTPSTDEZCTNVJNSDTQTPCYLRCTOP
```

kluc 2

```
VECDOXSORS CDYBSMUIMRCSPSOBXKQBSNO  
XGEFQZUQTUEFADUOWKOTEURUQDZMSDUPQ
```

kluc 3

VECDOXSORS CDYBSMUI MRCSPSOBXKQBSNO
YHF GRAVRUVFGBEVPXLPUFVSVREANTEVQR

... // dalsie kluce 4..26

Po prezretí všetkých možností by sme zistili že kľúč 16 sa dešifruje na „LUSTENIEHISTORICKYCHSIFIERNAGRIDE“.

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, SOMAR, HESLO, ...“.

kluc: JANO
SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW
JYXQJUZI JWMJXCJV FQA HQMKFJYEEIFCTN

kluc: SOMAR
SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW
AKYEBCYKSFHJUCFRAENEHYLTBG DGROXTK

kluc: HESLO
SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW
LUSTENIEHISTORICKYCHSIFIERNAGRIDE

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 sa ľahko môže dostať 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 procesy 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. Gén je zvyčajne 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 cyklu.

Účelová funkcia vypočítava skóre každého jedinca v populácii a je mierou toho, čo chceme maximalizovať, prípadne minimalizovať. Úlohou účelovej funkcie je nájsť globálny extrém.

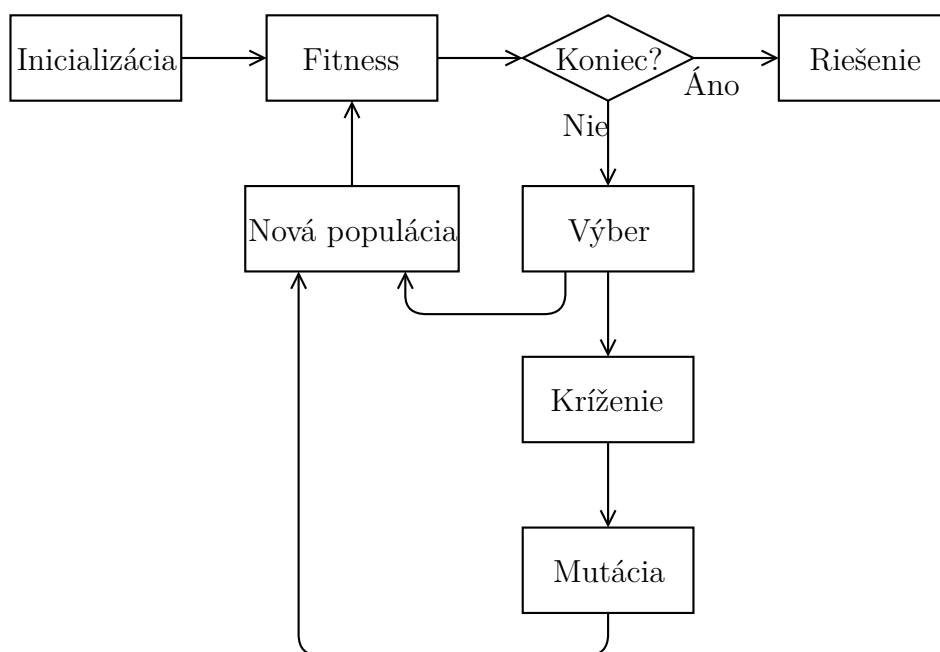
Fitness je v evolučných výpočtoch pojem predstavujúci mieru úspešnosti jedincov. V prípade maximalizačnej úlohy je to najväčšia hodnota účelovej funkcie. Naopak, v prípade minimalizačnej úlohy je to najmenšia hodnota.

Výber je proces, ktorý vyberie niektorých jedincov z populácie na základe zvolenej stratégie. Vybraný 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 si 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 definovať treba určit spôsob zakódovania parametrov jednotlivých objektov. Pri klasických šifrách sa gény zvyčajne reprezentujú pomocou abecedných znakov alebo čísiel. Ďalším krokom je určit prehľadávaný priestor a následne formulovať účelovú funkciu. Dôležité je aj určit veľkosť populácie, ktorá sa doporučuje medzi 10 a 100 jedincov [3]. Pri malých populácii býva nedostatočný priestor pre rôznorodosť (diverzitu) genetických informácii, naopak pri veľkých populáciách sa už nedosahuje lepšieho efektu.



Obrázok 1: Štruktúra GA

Princíp genetických algoritmov možno vidieť aj na obrázku 1, ktorý pozostáva

z niekoľkých krokov:

1. Prvým krokom je vygenerovanie počiatočnej populácie P_0 .
2. V tomto kroku sa vyhodnocuje populácia pomocou účelovej funkcie pre aktuálnu populáciu P_k .
3. V tejto fáze sa testuje ukončujúce podmienky genetického algoritmu. V prípade že algoritmus skončil vyberie za najlepšieho jedince aktuálnej populácie, ktorý predstavuje výsledné riešenie GA.
4. 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 vyberie iným spôsobom sú skopírované do operácie kríženia (skupina B).
5. V operácii kríženia sa náhodne spárujú rodičovské reťazce so skupiny B , s ktorých vznikne rovnaký počet potomkovských reťazcov. Potomkovia vytvoria skupinu C , ktorá pokračuje do operácie mutácie.
6. Mutácia spôsobí, že náhodne zvoleným jedincom zmenia niektoré náhodne zvolené gény a vznikne populácia D .
7. 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 vyzeráť, v skutočnosti však môžeme uvažovať o viacerých kritériách, ktoré vplývajú na GA. Niektoré uvedieme v experimentálnej časti tejto práce.

2 Grid

Jedným z cieľov 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, ktoré môžu ale nemusia byť geograficky oddelené. Účelom takýchto počítačov je poskytnúť veľký výpočtový výkon, ktorý je použitý na riešenie špecifických úloh.

2.1 hpc.stuba.sk

Vránci Slovenskej technickej univerzity (STU), Centra výpočtovej techniky (CVT) sa nachádza superpočítač IBM iDataPlex, ktorý pozostáva z 52 výpočtových uzlov. Každý výpočtový uzol má nasledovnú konfiguráciu:

- CPU: 2 x 6 jadrový Intel Xeon X5670 2.93 GHz
- RAM: 48GB (24GB na procesor)
- HDD: 2TB 7200 RPM 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 zdieľaného úložiska. Výpočtový výkon dosahuje 6,76 TFLOPS a maximálny príkon aj spolu s chladením je 40kW [4].

V tabuľke 1 môžeme vidieť dostupné diskové umiestnenia pre každého používateľa, prípadne úlohu. Umiestnenie `/home/$USER` je domovským priečinkom každého používateľa. Jedno z obmedzení tohto umiestnenia je, že môže obsahovať maximálne osemdesiat tisíc súborov a priečinkov. Taktiež má značné obmedzenú kapacitu čo sa nemusí hodiť pre každý typ úlohy. Ďalším umiestnením, ktoré ma používateľ k dispozícii je `/work/$USER`. Toto umiestnenie nemá žiadne väčšie obmedzenia slúži ako zdieľaný disk pre výpočty. Môžeme tu vytvárať ľubovoľný počet

súborov a priečinkov, avšak podľa [4] 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
<code>/home/\$USER</code>	áno	nie	32GB	80k inodes
<code>/scratch/\$PBS_JOBID</code>	nie	ihneď	1.6TB	nie
<code>/work/\$USER</code>	nie	áno	56TB	nie

Tabuľka 1: Disky

Aby sme boli schopný 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 môžeme prihlasovať bez zadávania hesla.

2.2 Príkazy

Do gridu sa môžeme prihlásiť cez SSH zadaním príkazu `ssh 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 obšiahlosť 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` nám deklaruje maximálny možný čas, ktorý má úloha PGA vyhradený. Hodnoty zo stĺpca `Job ID` môžeme použiť do príkazu `qsig` pre vynútené ukončenie úlohy.

2.2.3 qfree

Ak si chceme zobrazíť 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 [0] ... [0] 12.0 44.41 0 (0.0%) 0.00 0.00 job-exclusive
comp45 P [0] ... [0] 12.0 44.40 0 (0.0%) 0.00 0.00 job-exclusive
comp46 P [0] ... [0] 12.0 44.41 0 (0.0%) 0.00 0.00 job-exclusive
comp47 P [0] ... [0] 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 uzlu. 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, voľnú pamäť a využitie diskov. Posledný stĺpec **State** popisuje stav uzlu. 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, že uzol nie je dostupný, budú CPU označené ako [-].

2.3 Výpočtové fronty

Posledný a najdôležitejší príkaz je **qsub**, ktorý 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 preto sú spustené takmer okamžite. Debug fronta je obme-

dzená 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, na ktorej môžeme spúšťať jednoprocessorové 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

2.4 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. Poukážeme 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 prípade 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 akej bude úloha zaradená. V tomto prípade 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`.

2.5 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` (neposielat žiadny e-mail) [5].
- 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 dvanásť 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 stými, 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 je vysvetlený v kapitole mpi. .Pridat odkaz.

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.

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 laboratória a univerzity [6].

Úloha MPI fóra bola bola 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 vznikol v júni roku 1994 štandard 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 vyzeráť. 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 [7]. Pre ilustráciu si uvedme 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,
13                MPI_STATUS_IGNORE);
14        printf("%s\n", msg);
15    }
16    MPI_Finalize();
17    return 0;
18 }
```

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 príjemcu 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 bufferu**. 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.

3.2 Blokujúca komunikácia

Obe MPI volania v príklade 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 s rôzneho typu alebo rôzneho obsahu potrebujeme tieto správy nejako odlíšiť. Na odlíšenie správ slúži argument tag. Tag musí byť kladné celé číslo. V príklade 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` [7].

```
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 7. Štruktúra `MPI_Status` obsahuje 3 hlavné informácie: rank odosielateľa, tag správy a dĺžku správy.

3.3 Neblokujúca komunikácia

Výkonnosť programov môže byť v mnohých prípadoch vylepšená prekryvaním blokujúcich volaní. Jedným z možných spôsobov ako toto dosiahnuť je použitie threadov [7]. Alternatívny spôsob ako zlepšiť výkon programu môže byť použitie neblokujúcich komunikácií.

Pri neblokujúcej komunikácii si môžeme zadať 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`

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

complete, ktoré zaručí že dáta boli prenesené do recv buffera. Neblokujúca recv operácia podobne ako send pokračuje paralelne so zvyškom programu.

```

MPI_Request request[2];
if (rank == 0) {
    char msg1[] = "Hello, ", msg2[] = "MPI";
    MPI_Isend(msg1, strlen(msg1) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
              &request[0]);
    MPI_Isend(msg2, strlen(msg2) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
              &request[1]);
    zlozity_vypocet();
    MPI_Waitall(2, request, MPI_STATUSES_IGNORE);

} else if (rank == 1) {
    char msg1[20], msg2[20];
    MPI_Irecv(msg1, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request[0]);
    MPI_Irecv(msg2, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request[1]);

```

```

    iny_vypocet();
    MPI_Waitall(2, request, MPI_STATUSES_IGNORE);
    printf("%s%s\n", msg1, msg2);
}

```

Výpis 10: Neblokujúca komunikácia

Vo výpise 10 môžeme vidieť príklad neblokujúcej komunikácie. V tomto prípade 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 viac 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 operácii nie je zaručené. Správne poradie správ by sme docielili rôznymi tagmi pre správy `msg1` a `msg2`.

3.4 Dynamická alokácia

V príklade 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.

3.6 Boost MPI

Lepší spôsob poskytuje napríklad knižnica C++ Boost

```

#include "MpiApp.h"
int main(int argc, char **argv) {
    MpiApp app(argc, argv);
    if (app.rank() == 0) {
        app.send(1, 0, std::string("Hello MPI"));
    } else if (app.rank() == 1) {
        std::string msg;
        app.recv(0, 0, msg);
        std::cout << msg << std::endl;
    }
    return 0;
}

```

4 Návrh

Jedným zo základných cieľov tejto práce je poskytnúť jednoduchšie programové prostredie pre ďalších lúštitelov 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 súborami. Ďalším spoločným črtom 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.

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 podpriechniku `project`. Každý projekt po vytvorení obsahuje automaticky generované súbory: `CMakeLists.txt`, binárny spustiteľný súbor (po kompilácii), `.pbs` súbor a `main.cpp` v priechniku `src`. Do priechniku `src` by sa mali umiestňovať zdrojové súbory špecifické pre daný projekt. Ostatné zdrojové kódy, ktoré nesúvisia s daným projektom by mali ísť do modulov (priechinok `module`). Priechinok `module` slúži na zdieľanie zdrojových kódov (modulov) medzi projektami. Posledným priechinkom je `vendor`, ktorý slúži na uchovávanie knižníc od tretích strán.

4.2 Implementácia

V tejto práci sme si zvolili ako programovací jazyk `C++`, ktorý je dostatočne rýchly a poskytuje bohatú štandardnú knižnicu. Tento programovací jazyk sa tak-
tiež vyučuje na Fakulte elektrotechniky a informatiky a je veľmi rozšírený v indus-

triálnej sfére.

Keďže MPI štandard od verzie tri priamo nepodporuje C++ volania [7], 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.

Na správu zdrojových kódov sme použili systém `git` pretože archiváciu správy zdrojových kódov považujeme za veľmi dôležitú. Zdrojové kódy tohto projektu sú dostupné na githube <https://github.com/melias122/grid>.

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`. Taktiež skompiluje všetky potrebné závislosti. Kombinuje príkaz `module` z kapitoly (kap), a buildovací nástroj `cmake`.
- `-h, --help` argument vypíše návod ako pracovať s týmto príkazom.
- `-r, --run [projekt]` spustí projekt, ktorý je zadaný ako ďalší argument tohto príkazu. Hlavou myšlienkou tohto príkazu je že by sa mal správať podobne v lokálnom prostredí ale aj na gride. Príkaz parsuje `main.cpp` v priečinku projektu a podľa nastavenia `nodes`, `ppn`, `walltime` a `queue` dynamicky upravuje `.pbs` súbor pre grid, alebo tieto nastavenia použije do príkazu `mpirun` na simulovanie gridu.

- `-n`, `--new-project [nazov_projektu]` vytvorí projekt v priečinku `project`. Tento príkaz automaticky vytvára niektoré základné súbory ako `main.cpp`, a `.pbs` súbor.
- `--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, nezmažáva žiadne súbory.

4.3 Príklad použitia

Táto kapitola by mala slúžiť ako manuál pre ďalších používateľov. Postupne si prejdeme všetky kroky potrebné k použitiu nami vytvoreného softvéru.

4.3.1 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` [8].

Prvým krokom je nainštalovanie potrebných balíčkov príkazom `apt`.

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

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

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

V tretom kroku sa prepneme do repozitára a skompilujeme celý projekt. Ešte pred kompiláciou musíme vytvoriť súbor `.project-id`, ktorý by musí obsahovať identifikátor pre náš projekt. 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 jedna. Na vytvorenie nového projektu nám stačí zadať:

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

Čo vytvorí nasledovnú štruktúru v priečinku projekt:

```
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 neskompiluje. V prípade že by sme chceli pridať ďalší .cpp súbor, môžeme ho pridať do priečinku HelloGrid/src a následne ho musíme pridať do súboru HelloGrid/CMakeLists.txt.

```
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     cout << "Hello, grid from " << app.rank() << " out of " << app.size() <<
        endl;
26     return 0;
27 }

```

Výpis 12: HelloGrid/src/main.cpp

Zmeňme riadok šesť na `nodes = 2`, riadok trinásť na `walltime = 00:01:00` a riadok šesťnásť na `walltime = debug`. Parametre `nodes` a `ppn` sú dôležité pre spustenie programu na lokálnom počítači. Zvyšné parametre, `queue` a `walltime` sa prejaví až pri spustení na gride. Teraz môžeme projekt HelloGrid skompilovať a následne spustiť pomocou príkazov:

```

# skompiluje projekt HelloGrid a všetky závislosti
./build.sh -b

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

# vystup programu HelloGrid
Hello, grid from 2 out of 4
Hello, grid from 3 out of 4
Hello, grid from 0 out of 4
Hello, grid from 1 out of 4

```

Výstup programu nemusí vyzeráť presne takto.

4.3.2 Spustenie na gride hpc.stuba.sk

Aby sme boli schopný použiť program vytvorený v kapitole 4.3.1. Musíme najprv celý projekt skopírovať na grid hpc.stuba.sk. Možností ako skopírovať pro-

jekt je viacero. Môžeme ho skopírovať manuálne napríklad pomocou programu `rsync`, `scp` alebo môžeme použiť systém `git`, prípadne iný program. Pri manuálnom kopírovaní si treba dať pozor aby sme neskopírovali priečinky a súbory, ktoré závisia od lokálneho prostredia. Sú to hlavne súbory z predchádzajúcich kompilácií a priečinky `.build`, `vendor/boost`. Nachádzame sa v priečinku `/grid` a repozitáre môžeme zosynchronizovať pomocou:

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

Príkaz `./build.sh --sync` skopíruje celý projekt `/grid` 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 do fronty a skončení programu, 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:

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

Tentokrát, ak sme nespravili žiadnu zmenu na lokálnom pc, tak sa neskopíruje nič, ale stiahnu sa výstupy s programu `HelloGrid` do priečinku `project/HelloGrid`.

5 Experiment s GA

V tejto práci sme si zvolili genetické algoritmy pretože sú výpočtovo náročné a dajú sa paralelizovať. Tieto vlastnosti ich určujú ako vhodných kandidátov na nasadenie do gridového prostredia.

V našej práci sme sa rozhodli vyskúšať experiment s genetickými algoritmi, 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. Základnú implementáciu poskytol vedúci práce Ing. Eugen Antal, PhD. Táto implementácia bola ďalej nami rozvíjaná. Časť kódu bola prepísaná, ale základná idea ostala. Implementovali sme niektoré genetické operácie kríženia: single point crossover, uniform crossover.

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 iterácií bol 10000 a 50000. Zvolené schémy môžeme vidieť v tabuľke 4.

Prvým stĺpcom tabuľky je názov samotnej schémy. Ďalšie stĺpce tabuľky predstavujú zreteľenie 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 tretom 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 nový potomkovia nad ktorými sa vykoná operácia swap.

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 4: schemy

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 manhattanovskú vzdialenosť.

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 ktorom by sme mali jeden hlavný proces distribuujúci parametre n pracovných procesov. Tento spôsob môže mať dve riešenia. Prvým je, že hlavný proces rozhoduje komu posieľa 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 prijme 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 4. Pracovný proces sa snaží získať prácu od hlavného procesu tak, že mu najprv pošle svoje id a potom čaká kým mu hlavný proces prideli prácu (parametre), alebo zruší jeho činnosť.

```
void run_master(mpi::comm &comm) {  
    for (int itteration : {10000, 50000} ) {  
        for (int populationSize : {10, 20, 50 , 100}) {
```

```

    for (int textsize = 50; textsize <= 2000; textsize += 50) {
        for (int text = 1; text <= 100; text++) {
            string pt = read_plaintext();
            string ct = read_ciphertext();
            Data data;
            comm.recv(mpi::any_source, 0, data);
            // copy data
            data.itteration = itteration;
            ...
            comm.send(data.workerId, 1, data);
        }
    }
}
// shutdown workers
}

void run_worker(int id, mpi::comm &comm) {
    // initialize
    while (1) {
        Data data(id);
        // nonblocking send
        comm.isend(master, 0, data);
        comm.recv(master, 1, data);
        if (!data.status) return;
        // setup GA scheme
        Scheme scheme = ...
        for (schemeId &s : {A, B, C, ...}) {
            GeneticAlgorithm::run(scheme);
            // write best
        }
    }
}

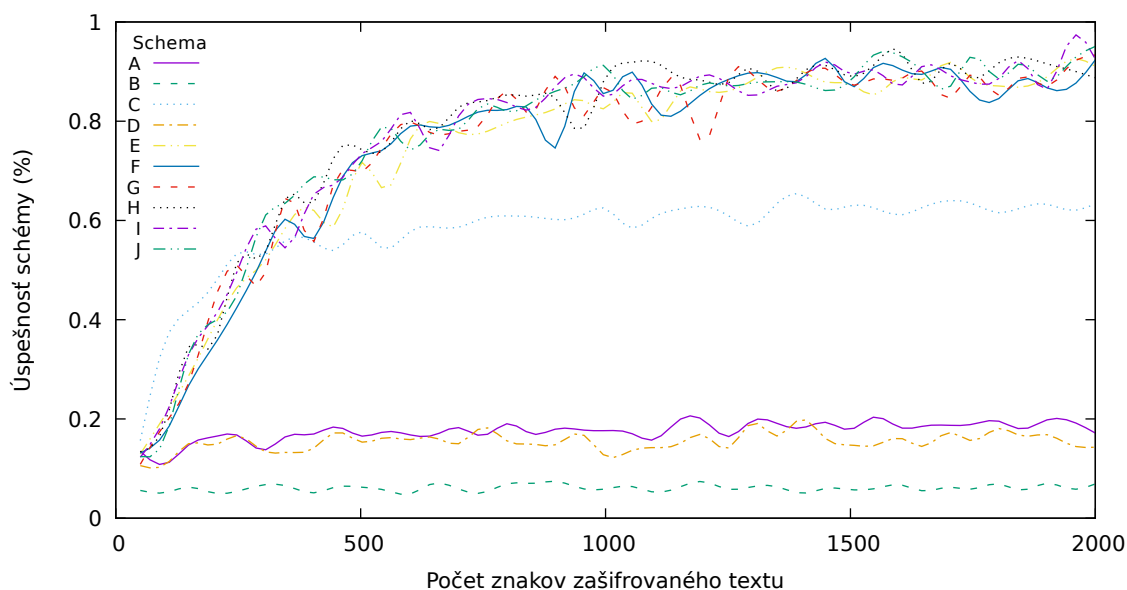
```

Výpis 13: Pseudokód distribúcie parametrov GA

5.2 Výsledky

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 2 môžeme vidieť jeden z výsledkov genetického algoritmu. Ako možno vidieť z grafu najhoršie výsledky dosiahla schéma B, ktorá je zložená z náhodného výberu a jedeného swapu. Táto schéma sa správa podľa našich očakavaní. 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, u 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.



Obrázok 2: Počet iterácií: 10000, počiatočná populácia: 10

Na ďalšom obrázku 3 môžeme vidieť GA s počiatočnou populáciou o veľkosti 50 jedincov. Zvýšenie počtu jedincov spôsobuje malé zlepšie „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 elitný jedincov, dosahujú z vyšším počtom jedincov mierne zlepšenie u kratších textov.

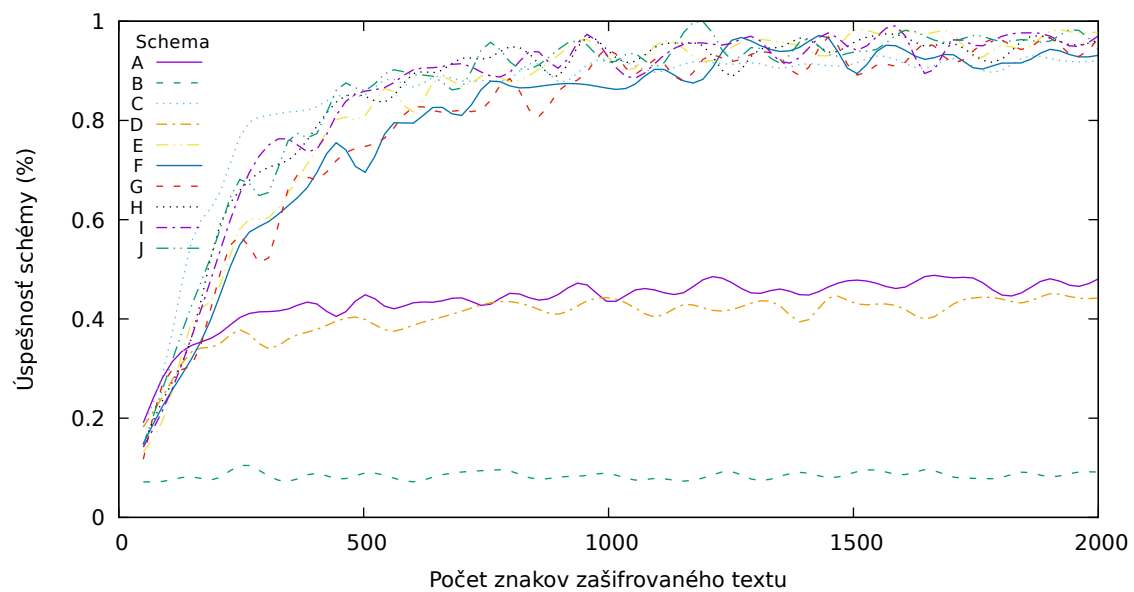


Obrázok 3: Počet iterácií: 10000, počiatočná populácia: 50

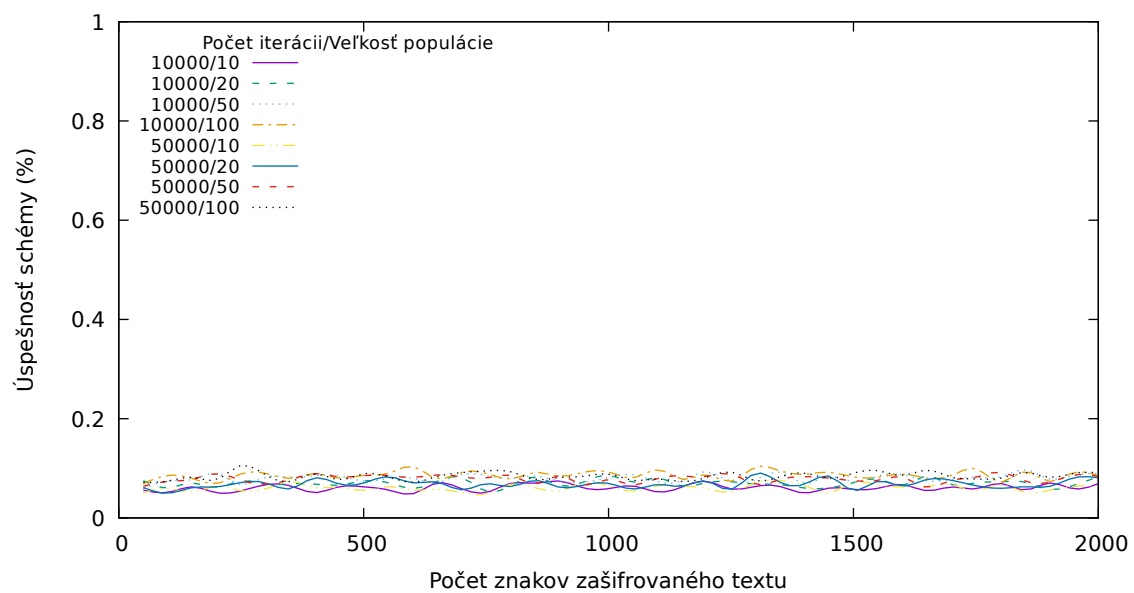
Podľa očakávaní našich očakávaní boli dosiahnuté najlpšie výsledky s nastavením genetického algoritmu pri 50000 iterácii a s počiatočnou populáciou veľkosti 100 jedincov (obrázok 4). Lepšie výsledky boli dosiahnuté najmä pri kratších textoch a mierne zlepšenie môžeme vidieť aj pri dlhších textoch. Treba však poznamenať, že GA pri takom nastavení beží značne dlhšie a dobré výsledky sa dajú dosiahnuť aj s menším počtom iterácií a menšou populáciou.

Zo schémy B na obrázku 5 možno vidieť, že náhodne zvolený jedinci prinášajú do GA diverzitu.

Ako sme už poznamenali schéma C (obrázok 6) 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

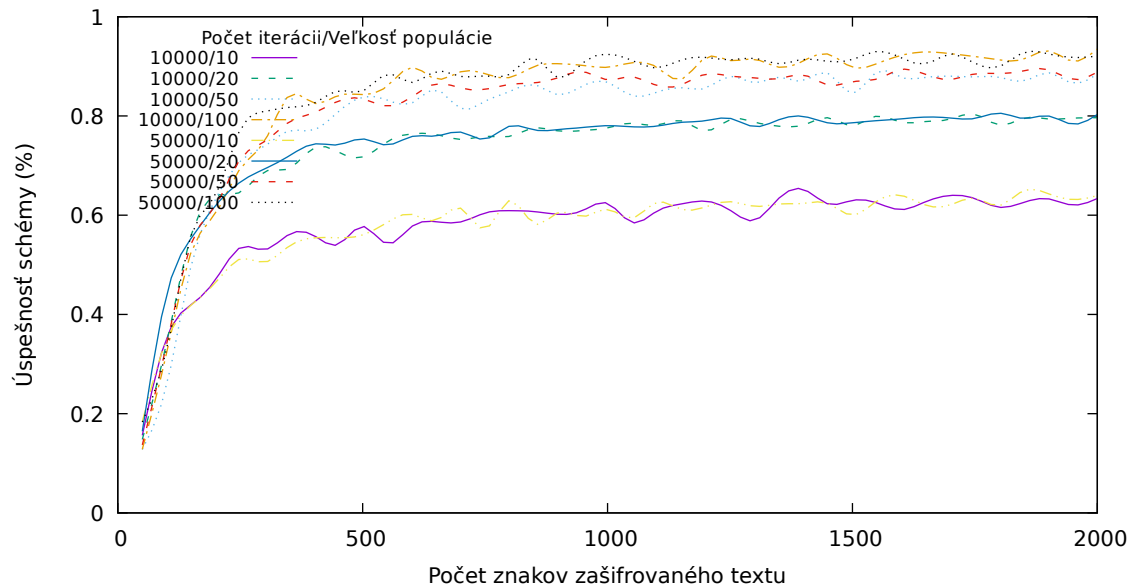


Obrázok 4: Počet iterácií: 50000, počiatočná populácia: 100



Obrázok 5: Schéma: B

tejto schémy napríklad zväčšenie populácie z 50 a 100 jedincov už neprinieslo o tolko lepšie výsledky a je možné že príliš veľká populácia by tejto schéme mohla aj uškodiť pôsobením väčšej rôznorodosti.



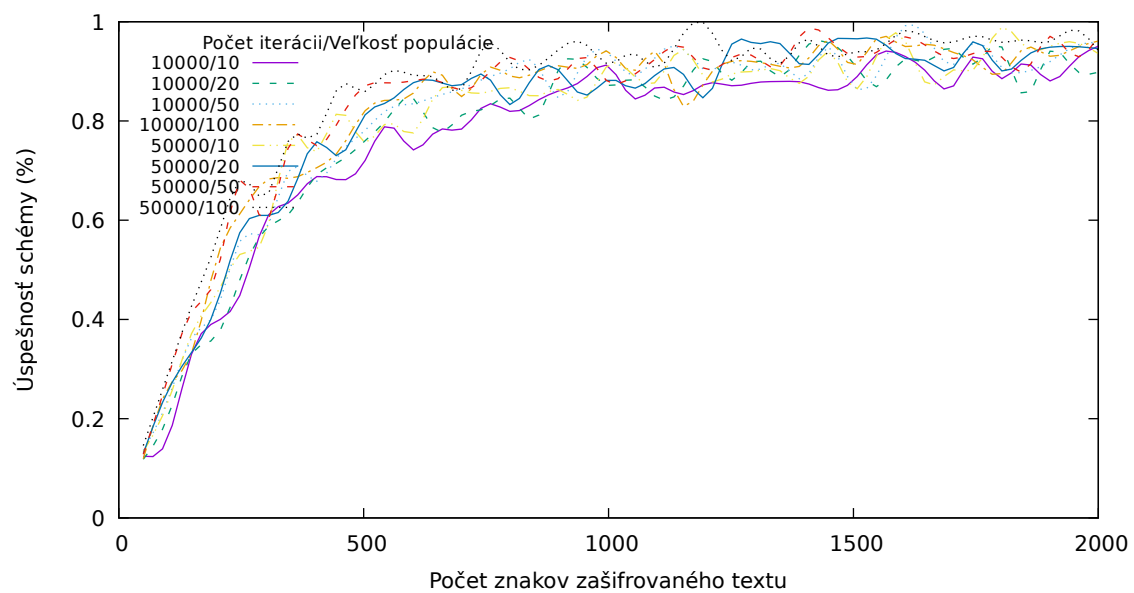
Obrázok 6: Schéma: C

Na ďalších obrázkoch schém E (obrázok 7) a F (obrázok B.8), môžeme vidieť že zväčšenie počtu iterácií GA ako aj veľkosti populácie prináša pri lúštení monoalfabetickej substitúcie iba mierne zlepšenie úspešnosti. Z grafov týchto funkcií že zlepšenie sa pohybuje okolo 10%.

Zvyšné výsledky z tohto experimentu môžeme vidieť v prílohe B tejto práce.



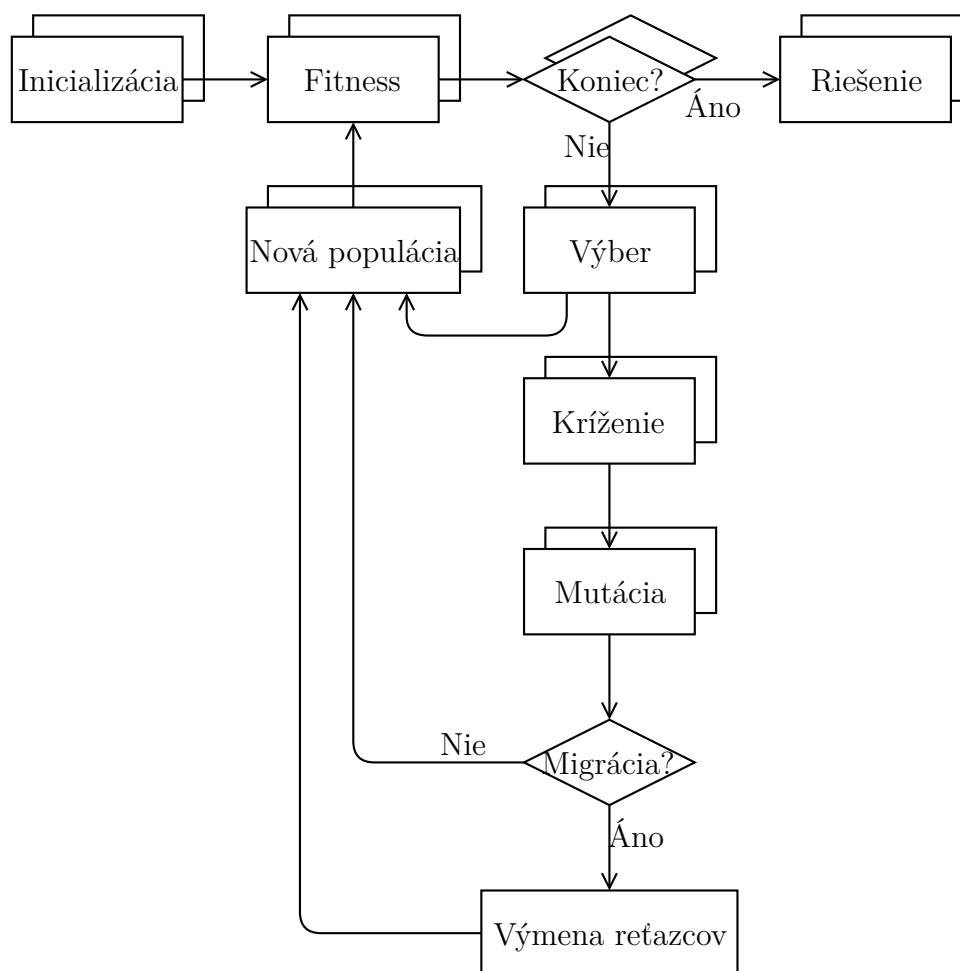
Obrázok 7: Schéma: E



Obrázok 8: Schéma: J

6 Experiment s PGA

Paralelné genetické algoritmy (PGA) možno chápať ako ďalšiu úroveň paralelnej organizácie populácie, ktorá ma za cieľ zýšiť výkonnosť aj efektívnosť genetických algoritmov [9]. PGA prinášajú výhody v porovnaní s GA, ktoré sú podľa [9] sú: paralelné hľadanie vo veľkom mnohodoménovom priestore potenciálnych riešení, lepšia možnosť vnútornej organizácie v porovnaní s GA, vyššia efektívnosť pri rovnakom počte vyhodnotení účelovej funkcie v porovnaní s GA, schopnosť nájsť lepšie riešenia, schopnosť uvoľniť sa zo stagnácie v lokálnom extréme a smerovať ku globálnemu extrému.



Obrázok 9: Štruktúra PGA s migračným modelom

Na obrázku 9 môžeme vidieť zjednodušenú blokovú shému PGA s migračným modelom. Pri takomto type prebieha výpočet jednotlivých genetických algoritmov vo viacerých nezávislých, ale spolupracujúcich subpopuláciách (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.

Štruktúra topológie medzi populáciami môže byť rôzna. Jednotlivé populácie medzi sebou nemusia byť všetky previazané. Niektoré vybrané topológie z [9] môžeme vidieť na obrázku 10. 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 jedna veľká populácia.

6.1 MpiMigrator

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

Vytváranie topológií sa nám podarilo zovšeobecniť nasledovným spôsobom:

```
class Migrator {
public:
    Migrator(int migrationTime);
    virtual void migrate(int senderId, int iteration, Population &population) =
        0;
    // sender -> receiver
    void addMigration(int senderId, int receiverId, Migration::Type t, int n)
    // ...
};
```

Migrator je základná 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 môže byť napríklad: najlepší, najhorší, nahodný a iné. Pre príjemcu samozrejme

platí, že očakáva migráciu takéhoto typu. Funkciu `migrate` sa musí implementovať zvlášť pre kontrétny PGA Migrator. Konkrétne implementácie môžu byť pre: `thready`, `MPI`, `OpenMP`, a ďalšie.

V našom prípade sme implementovali `MPI` migrátor. Migrátor pošle niektorých jedincov populácie vopred zadaného typu. Posielanie nesmie byť blokujúce, pretože to by znamenalo deadlock. Prijatie môže byť blokujúce odosielateľ príjme nových jedincov ak existuje k nemu priradená väzba. Prijatých jedincov potom zaradi do aktuálnej populácie a nakoniec čaká na dokončenie všetkých neblokujúcich volaní.

```
// nonblocking send
for (const Migration &m : p_senderMigrations[senderId]) {
    Population pop = m.select(p);
    mpi::request r = comm.isend<Population>(m.receiverId(), 0, pop);
}

// recv
for (const int &id : p_receiverMigrations[senderId]) {
    Population pop;
    comm.recv<Population>(id, 0, pop);
    append(p, pop);
}

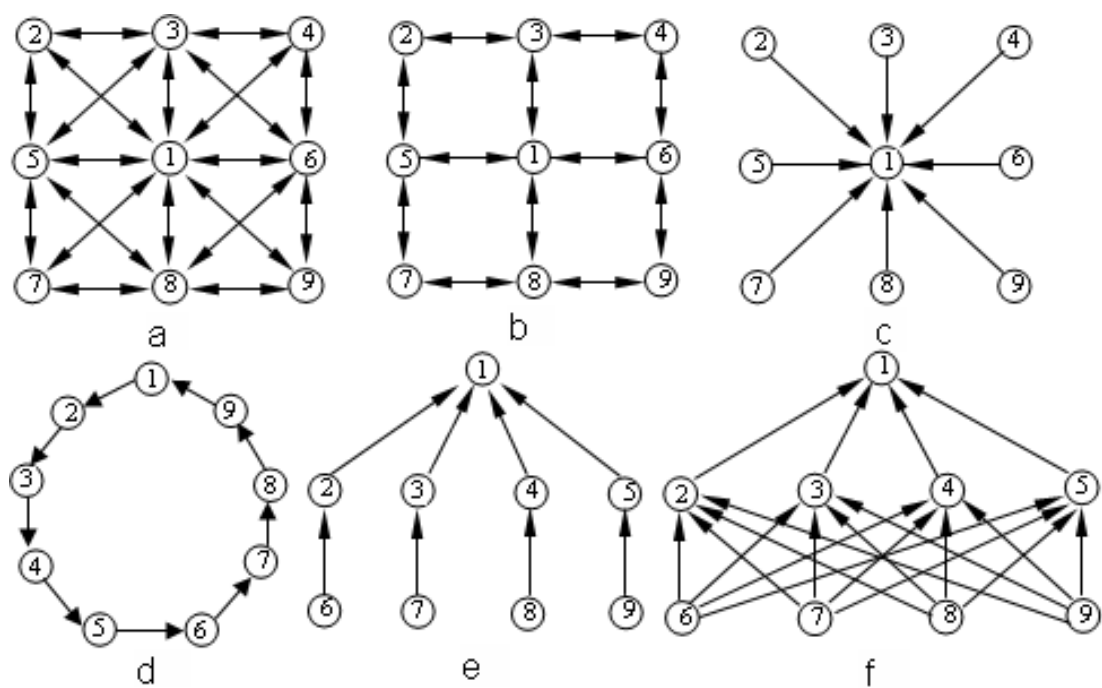
// wait for non blocking send
for (auto &r : request) {
    r.wait();
}
```

Výpis 14: Pseudokód `MpiMigrator`

6.2 Topológia

6.3 Výsledky experimentu

Výsledky PGA a ich interpretácia ...



Obrázok 10: Topológie paralelných genetických algoritmov

Záver

Conclusion is going to be where?

Here.

Zoznam použitej literatúry

1. GROŠEK, O., VOJVODA, M. a ZAJAC, P. *Klasické šifry*. Slovenská technická univerzita, 2007. ISBN 978-80-227-2653-5.
2. 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.
3. SEKAJ, I. *Evolučné výpočty a ich využitie v praxi*. Iris, 2005. ISBN 9788089018871.
4. *STUBA klaster - IBM iDataPlex*. Dostupné tiež z: <https://www.hpc.stuba.sk>.
5. *qsub*. Adaptive Computing, 2012. Dostupné tiež z: <http://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm>.
6. 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.
7. FORUM, Message Passing Interface. *MPI: A Message-Passing Interface Standard*. 2015. Dostupné tiež z: <http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.
8. *Bash on ubuntu on Windows*. 2017. Dostupné tiež z: <https://msdn.microsoft.com/commandline/wsl/about>.
9. IVAN SEKAJ, Michal Oravec. Paralelné evolučné algoritmy.

Prílohy

A	Štruktúra elektronického nosiča	II
B	Výsledky Genetického algoritmu	III

A Štruktúra elektronického nosiča

/CHANGELOG.md

- file describing changes made to FEIstyle

/example.tex

- main example *.tex* file for diploma thesis

/example_paper.tex

- example *.tex* file for seminar paper

/Makefile

- simply Makefile – build system

/fei.sublime-project

- is project file with build in Build System for Sublime Text 3

/img

- folder with images

/includes

- files with content

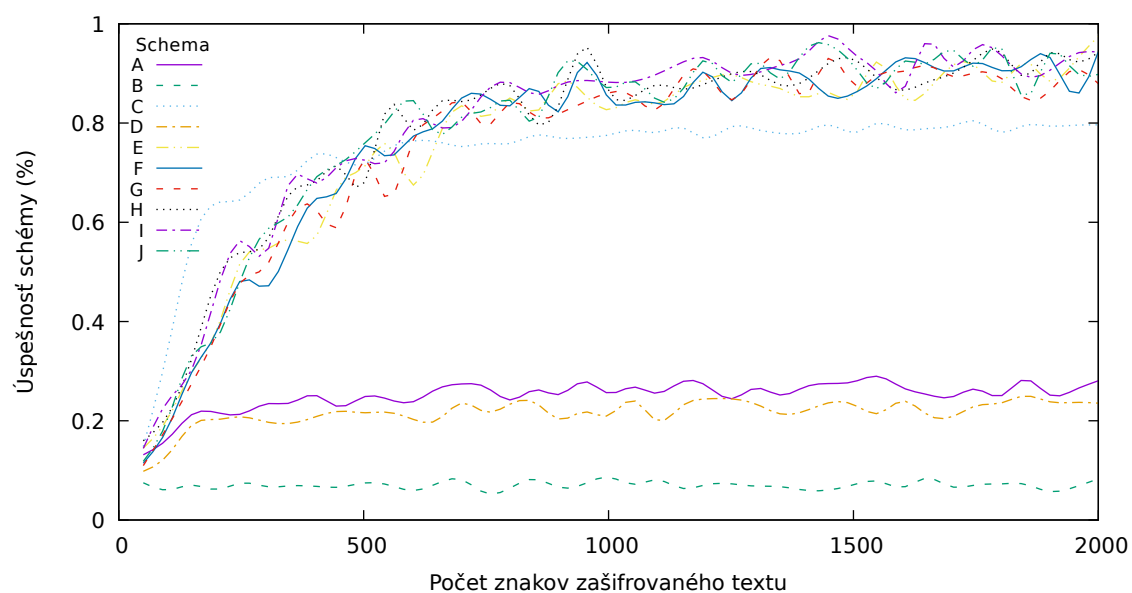
/bibliography.bib

- bibliography file

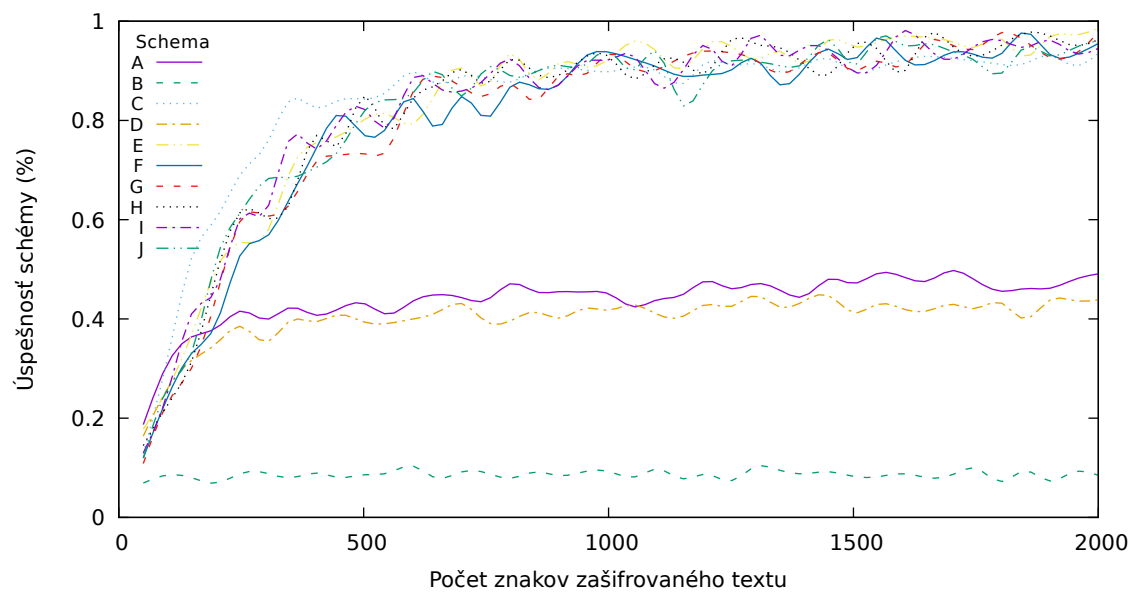
/attachmentA.tex

- this very file

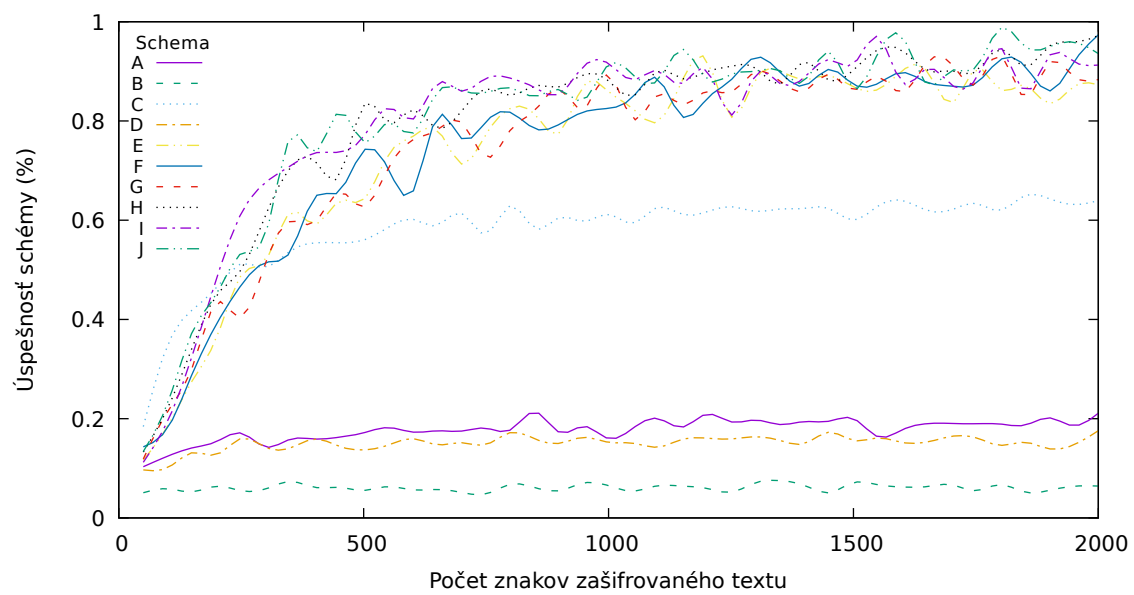
B Výsledky Genetického algoritmu



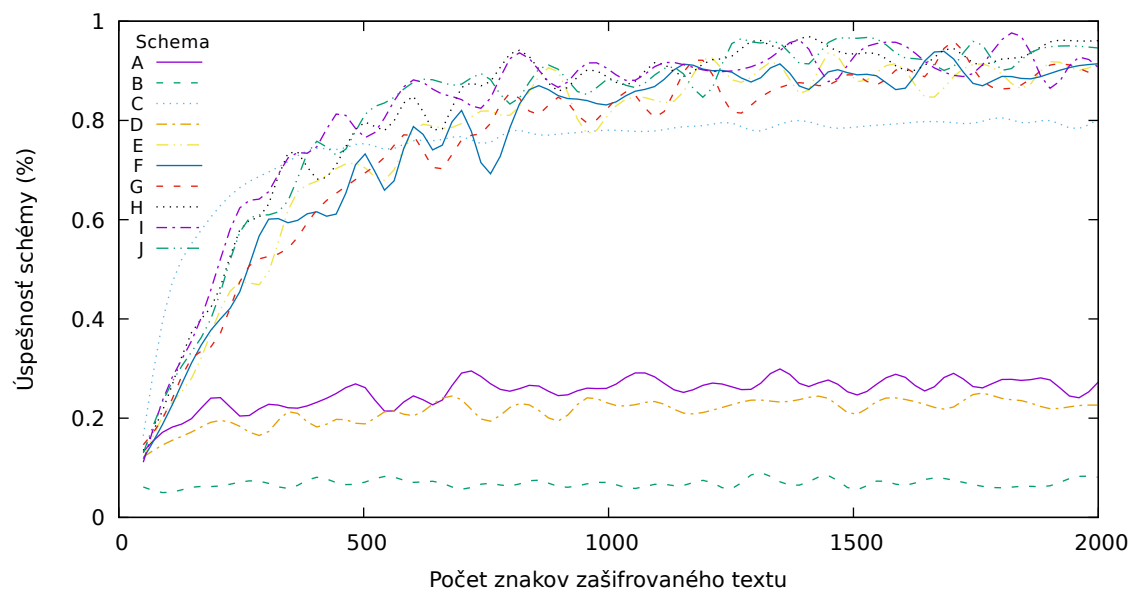
Obrázok B.1: Počet iterácií: 10000, počiatočná populácia: 20



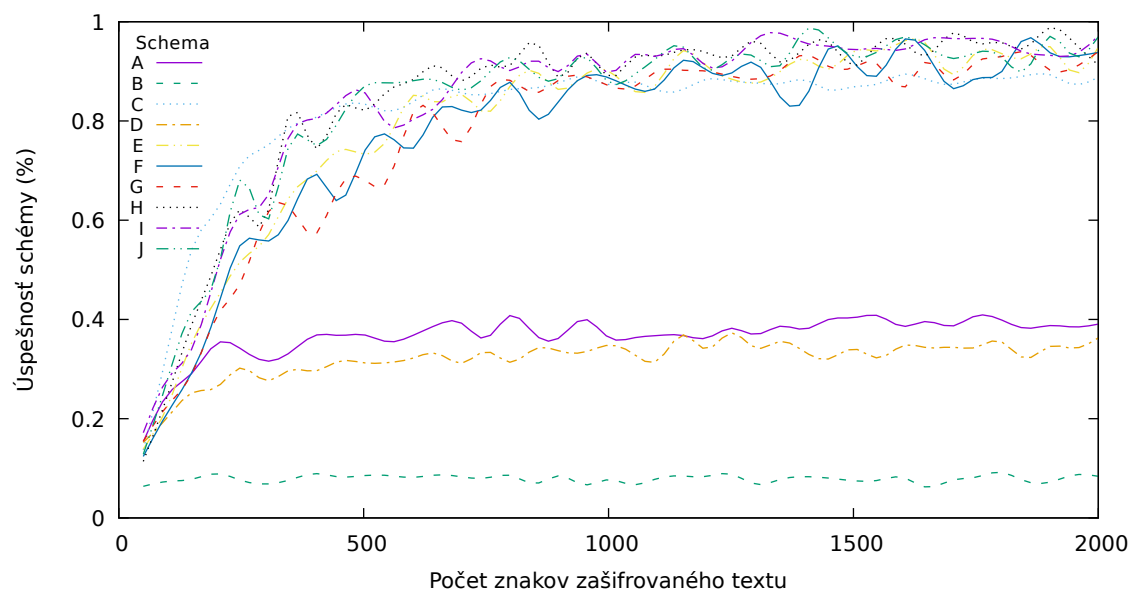
Obrázok B.2: Počet iterácií: 10000, počiatočná populácia: 100



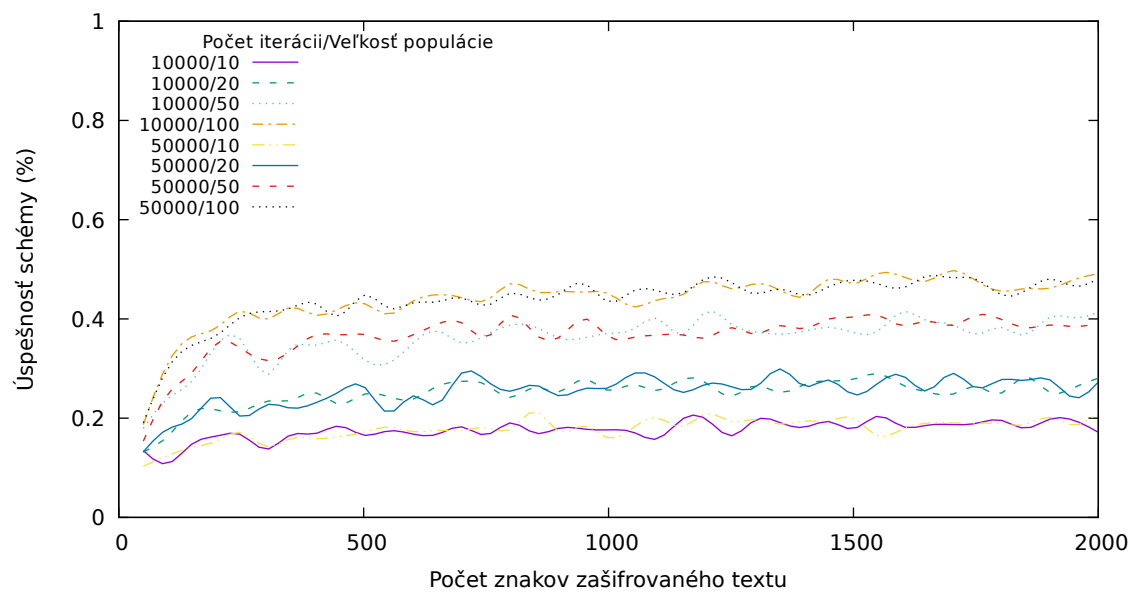
Obrázok B.3: Počet iterácií: 50000, počiatočná populácia: 10



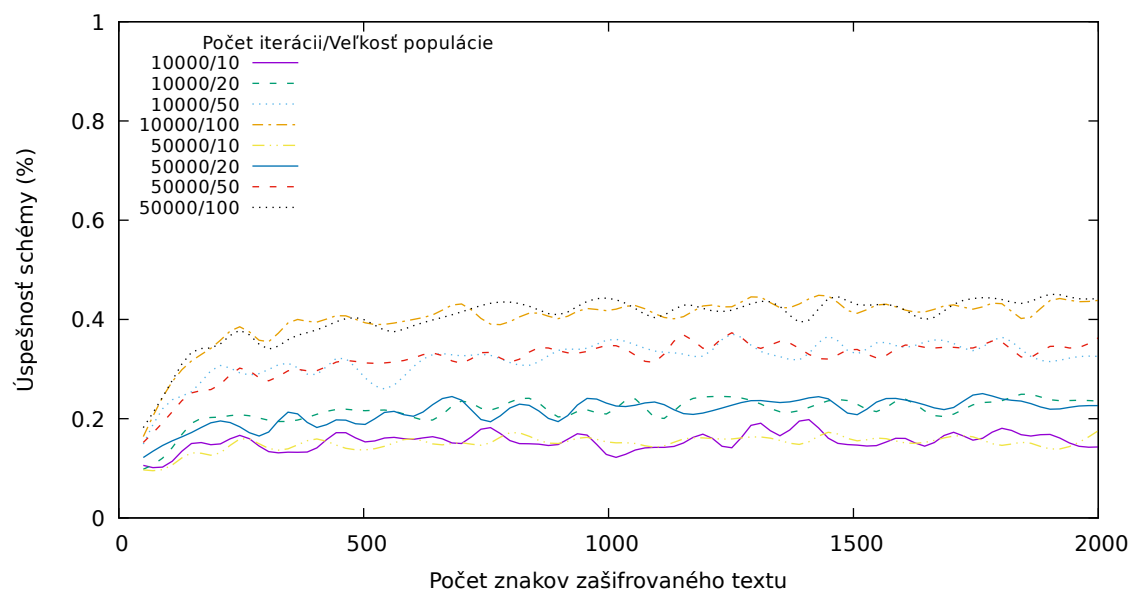
Obrázok B.4: Počet iterácií: 50000, počiatočná populácia: 20



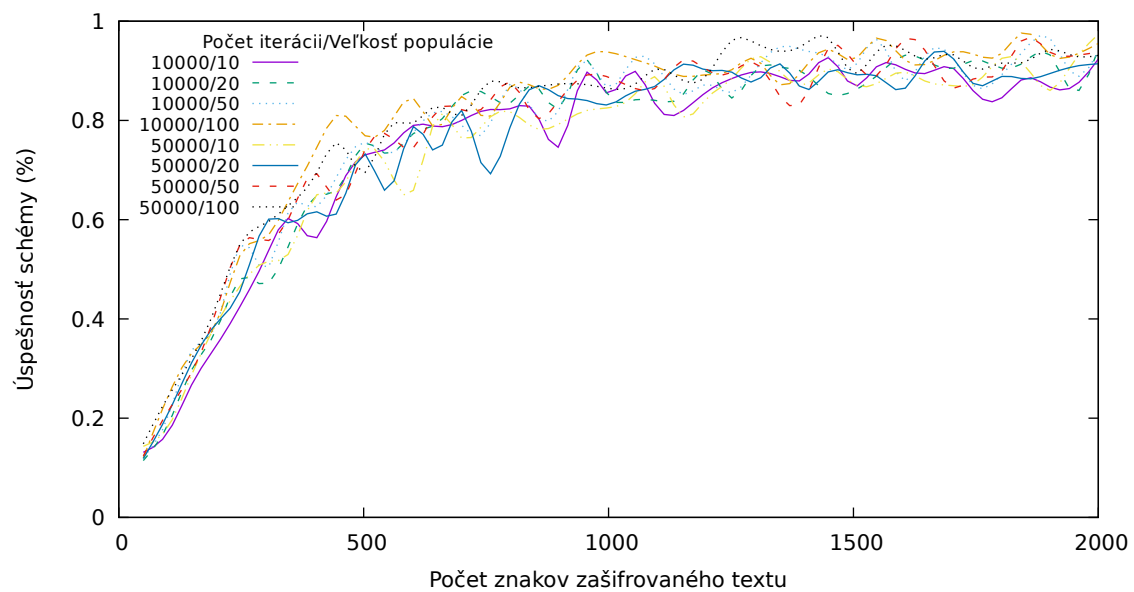
Obrázok B.5: Počet iterácií: 50000, počiatočná populácia: 50



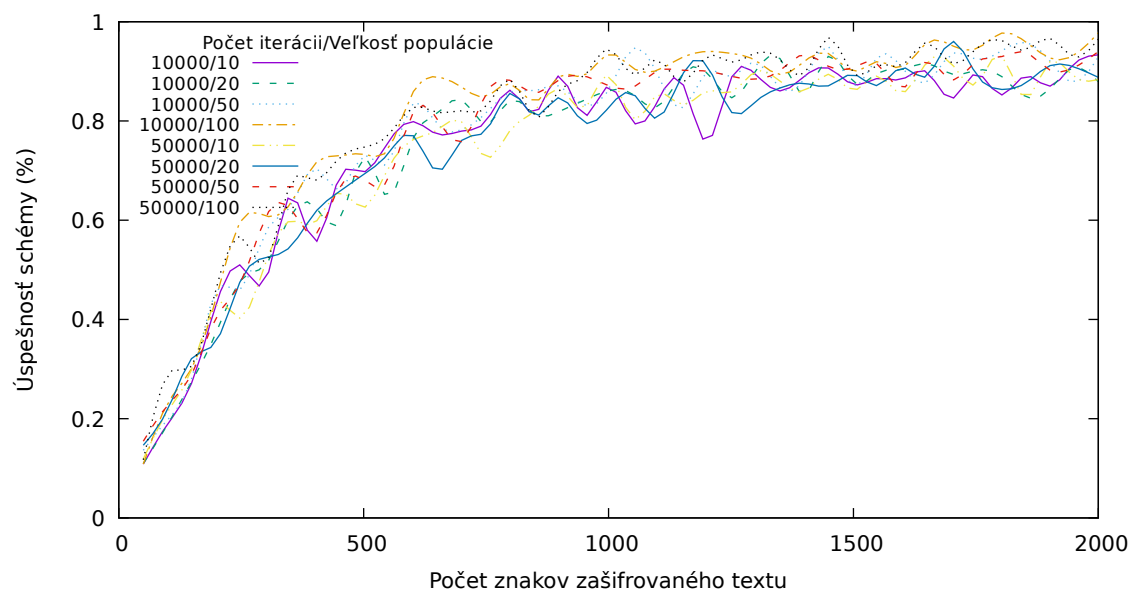
Obrázok B.6: Schéma: A



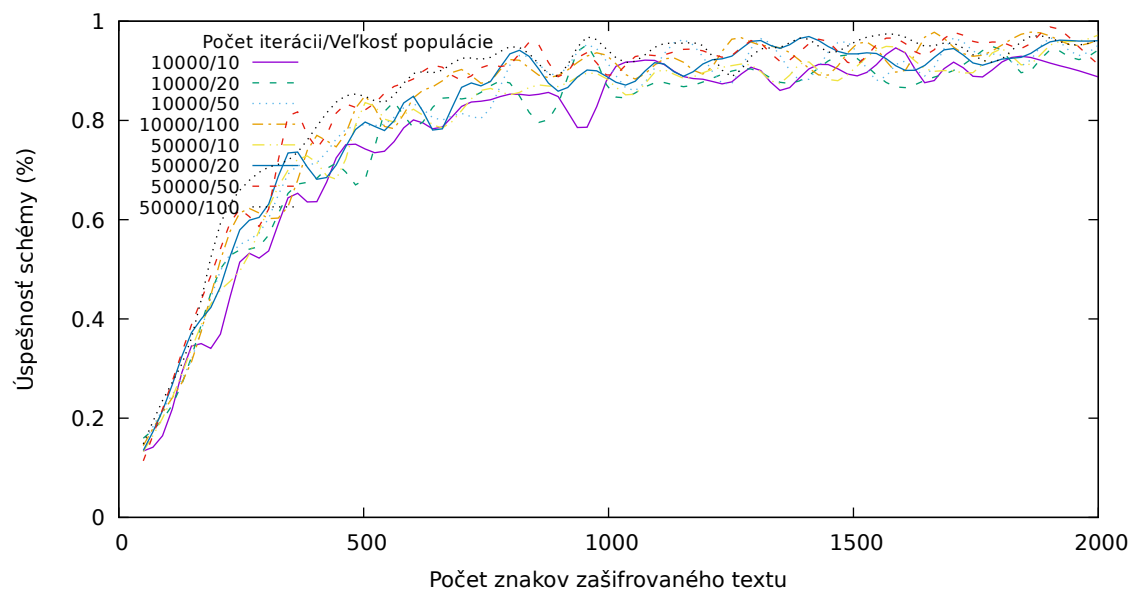
Obrázok B.7: Schéma: D



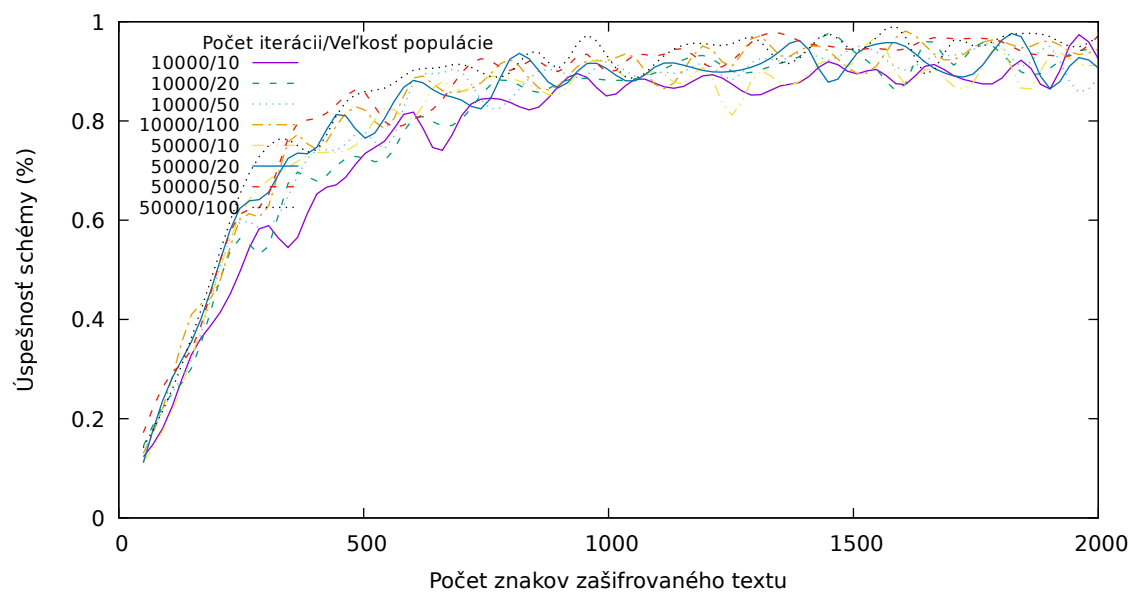
Obrázok B.8: Schéma: F



Obrázok B.9: Schéma: G



Obrázok B.10: Schéma: H



Obrázok B.11: Schéma: I