

**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 Útoky	5
1.3.1 Hrubou silou	5
1.3.2 Slovníkový útok	6
1.3.3 Genetické a evolučné algoritmy	6
2 Grid	7
2.1 hpc.stuba.sk	7
2.2 Príkazy	8
2.2.1 module	9
2.2.2 qstat	9
2.2.3 qfree	10
2.3 Výpočtové fronty	11
2.4 Príklad sériovej úlohy	12
2.5 Príklad paralelnej úlohy	13
3 MPI	16
3.1 Point-to-point komunikácia	16
3.2 Blokujúca komunikácia	18
3.3 Neblokujúca komunikácia	19
3.4 Dynamická alokácia	21
3.5 Serializácia dátových typov	22
3.6 Boost MPI	22
4 Návrh	23
4.1 Štruktúra prostredia	23
4.2 Príklad použitia	24

4.2.1	Lokálny vývoj	24
4.2.2	Použitie na gride hpc.stuba.sk	27
Záver		29
Zoznam použitej literatúry		30
Prílohy		I
A Štruktúra elektronického nosiča		II
B Výsledky Genetikého algoritmu		III

Zoznam obrázkov a tabuliek

Obrázok B.1	Počet iterácii: 10000, počiatočná populácia: 10	III
Obrázok B.2	Počet iterácii: 10000, počiatočná populácia: 20	IV
Obrázok B.3	Počet iterácii: 10000, počiatočná populácia: 50	V
Obrázok B.4	Počet iterácii: 10000, počiatočná populácia: 100	VI
Obrázok B.5	Počet iterácii: 50000, počiatočná populácia: 10	VII
Obrázok B.6	Počet iterácii: 50000, počiatočná populácia: 20	VIII
Obrázok B.7	Počet iterácii: 50000, počiatočná populácia: 50	IX
Obrázok B.8	Počet iterácii: 50000, počiatočná populácia: 100	X
Tabuľka 1	Disky	8
Tabuľka 2	Výpočtové fronty a ich obmedzenia	12
Tabuľka 3	Dátové typy v MPI a ich C ekvivalenty	20

Zoznam skratiek

BASH	Bourne Again SHell
CPU	Central processing unit
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	9
2	qstat	9
3	qstat -u 3xelias	10
4	qfree	10
5	uloha1.pbs	12
6	uloha2.pbs	13
7	Point-to-point komunikácia	17
8	MPI_Send	18
9	MPI_Recv	19
10	Neblokujúca komunikácia	20
11	Štruktúra prostredia	23
12	HelloGrid/src/main.cpp	26

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 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 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 1917 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.

Z 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 operácii.

- **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 blok otvoreného textu sa 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 Útoky

1.3.1 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 „VECDXSORS CDYBSMUIMRCSPSOBXKQBSNO“, ktorý vieme že bol zašifrovaný šifrou podobnou Cézarovkej šifre. Pre získanie otvoreného textu potrebujeme vyskúšať všetkých 26 možností posunov, čo je v tomto prípade kľúč, tak aby sme dostali zmysluplný text.

klúč 1

VECDXSORS CDYBSMUIMRCSPSOBXKQBSNO
WFDEPYTPSTDEZCTNVJNSDTQTPCYLRCTOP

klúč 2

VECDXSORS CDYBSMUIMRCSPSOBXKQBSNO
XGEFQZUQTUEFADUOWKOTEURUQDZMSDUPQ

klúč 3

VECDXSORS CDYBSMUIMRCSPSOBXKQBSNO
YHFGRAVRUVFGBEVPXLPUFVSVREANTEVQR

... // ďalšie kľúče 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, ...“.

klúč JANO

SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW
JYXQJUZI JWMJXCJVFQAHQMKFJYEEIFCTN

klúč SOMAR

SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW
AKYEBCYKSFHJUCFRAENEHYLTBGDGROXTK

klúč HESLO

SYKESUMWSWZXGCWJOQNVZMXTSYRSRFPHW
LUSTENIEHISTORICKYCHSIFIERNAGRIDE

1.3.3 Genetické a evolučné algoritmy

todo

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

V rámci 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 [3].

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

súborov a priečinkov, avšak podľa [3] 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 na vyťaženie 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 [-].

Posledný a najdôležitejší príkaz je `qsub`, ktorý slúži na zaradenie úloh, PBS skriptov do výpočtovej fronty.

2.3 Výpočtové fronty

Aby sme boli schopný 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 odľadenie úloh. Úlohy v tejto fronte

majú vysokú prioritu 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, na ktorej 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

2.4 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 ktorý sériovej úlohy. Popíšme si 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 si v tomto príklade 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 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) [4].
- 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 premmenných 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, ktorých 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 odelených 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 devädesiatich 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 štyridsať rôznych organizácií, ktoré predstavovali predajcov paralelných systémov, používateľov ale aj výskumné laboratória a univerzity [5].

Úloha MPI fóra 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 prímanie správ je kľúčovým stavebným mechanizmom MPI komunikácie. Základné operácie sú **send** (posielanie) a **receive** (prímanie). Výmene

správ medzi dvoma procesmi hovoríme *point-to-point* komunikácia. Takmer všetky MPI konštrukcie sú založené na point-to-point komunikácii [6]. 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 tazkvaný **send buffer**. Send buffer je miesto v pamäti odosielaťa, v ktorom sa nachádzajú dáta na odoslanie. V tomto príklade send buffer 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. Ďalsie tri parametre send operácie sú definujú príjemcu správy. Proces jedna (`rank == 1`), príjme 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. Ďalsie tri parametre slúžia na zvolenie správy od odosielaťa. 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 ze proces jedna vypíše Hello, MPI.

3.2 Blokujúca komunikácia

Obe MPI volania v príklade 7 boli obe blokujúce. To znamená, že odosielateľ čaká na volaní `MPI_Send` dovtedy až 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 vymenávajú viacej správ s rôznym typom 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 vymenávať správy. MPI štandard

zahŕňa aj vytváranie vlastných podskupín komunikátorov, napríklad pomocou volania `MPI_Comm_create_group` [6].

```
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 indentické. Argument `source` v prípade funkcie `MPI_Recv` definuje rank odosielateľa. Dalším 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 prekrívaním blokujúcich volaní. Jedným z možných spôsobov ako toto dosiahnuť je použitie threadov [6]. Alternatívny spôsob ako zlepšiť výkon programu môže byť použitie neblokujucech komunikácie.

Pri neblokujucej komunikácii si môžeme zadať štyri rôzne neblokujuce operácie: `send` pre posielanie, `recv` pre prímanie, `send complete` pre dokončenie posielania a `recv complete` pre dokončenie prijatia. Neblokujuce volanie `send` spustí send operáciu, ale nedokončí ju. Toto volanie inicializuje skopírovanie do send buffera ale samotné kopírovanie nedokončí. Na dokončenie operácie `send` je potrebná ďalšie volanie `send complete`, ktorá overí že dáta boli skutočne skopírované a prenesené príjemcovi. Neblokujuca send operácia zvyčajne beží paralelne s vykonávaným programom. Podobne aj neblokujuce 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 neblokujúcich 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 kým správy neboli prenesené. 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 vidíme 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 „Hello, MPI“ alebo aj „MPIHello, “, pretože poradie vykonania operácii nie je zaručené. Správneme poradie správ by sme dosielili 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. Aby sme mohli dynamicky alokovať pamäť 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 vymenovať 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 aj 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, ktorý budú pri svojej práci využívať grid. Takéto prostredie by malo poskytnúť spoločné stavebné bloky, ktoré by si riešitelia medzi sebou mohli navzájom zdieľať.

V tejto práci sme si zvolili ako programovací jazyk C++, ktorý je dostatočne rýchly a poskytuje bohatú štandardnú knižnicu. Keďže MPI štandard od verzie tri priamo nepodporuje C++ volania [6], budeme používať knižnicu Boost a jej MPI nadstavbu. Z Boost knižnice využijeme aj module serialization pre dátovú serializáciu.

Na správu zdrojových kódov sme použili systém `git`. čo je to, preco sme to použili? Ako buildovací systém sme použili `cmake`.

4.1 Štruktúra prostredia

V tejto kapitole popíšeme 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

Základom 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.

Skript `build.sh` vytvára projekty do podpriechinky `project`. Každý projekt po vytvorení obsahuje automaticky generované súbory: `CMakeLists.txt`, binárny spustiteľný súbor, (po kompilácii), `.pbs` súbor a `main.cpp` v priechinke `src`. Do priechinky `src` by sa mali umiestovať 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 Príklad použitia

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

4.2.1 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 Win-

dows 10, ktorý v najnovších verziách podporuje `bash` a balíčkový systém `apt` [7].

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.

```
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. Prvý krát 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`. V použitia tohto príkazu musíme pokračovať od kroku tri. Na vytvorenie nového projektu nam staci zadať:

```
# vytvorenie nového projektu s názvom 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 cheli pridať ďalší `.cpp` súbor,

môžeme ho pridať priečinku `HelloGrid/CMakeLists.txt` a následne ho pridať do súboru `HelloGrid/CMakeLists.txt`.

```
1 // Project HelloGrid
2 //
3 // nodes defines number of nodes to use
4 // ppn defines number of procesors 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 šestnásť na `walltime = debug`. Parametre `nodes` a `ppn` sú dôležité pre spustenie programu na lokálnom počítači. Zvyšné parametre, `queue` a `walltime` sa prejavujú 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.

4.2.2 Použitie na gride hpc.stuba.sk

Aby sme boli schopný použiť program vytvorený v kapitole 4.2.1. 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. Pri manuálnom kopírovaní si treba dať pozor aby sme neskopírovali priečinky a súbory, ktoré môžu mať závislosti na lokálne prostredie. 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 nechcené súbory. Po skopírovaní súborov sa môžeme prihlásiť pomocou `ssh`.

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

# prepnutie sa do priečinku 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 prípadе nevidíme 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 priečinku ~/grid  
./build.sh --sync
```

Tentokrát ak sme nespravili žiadnu zmenu na lokalnom pc, tak sa neskopiruje nič, ale stiahnu sa výstupy s programu HelloGrid.

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. *STUBA klaster - IBM iDataPlex*. Dostupné tiež z: <https://www.hpc.stuba.sk>.
4. *qsub*. Adaptive Computing, 2012. Dostupné tiež z: <http://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm>.
5. 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.
6. 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>.
7. *Bash on ubuntu on Windows*. 2017. Dostupné tiež z: <https://msdn.microsoft.com/commandline/wsl/about>.

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

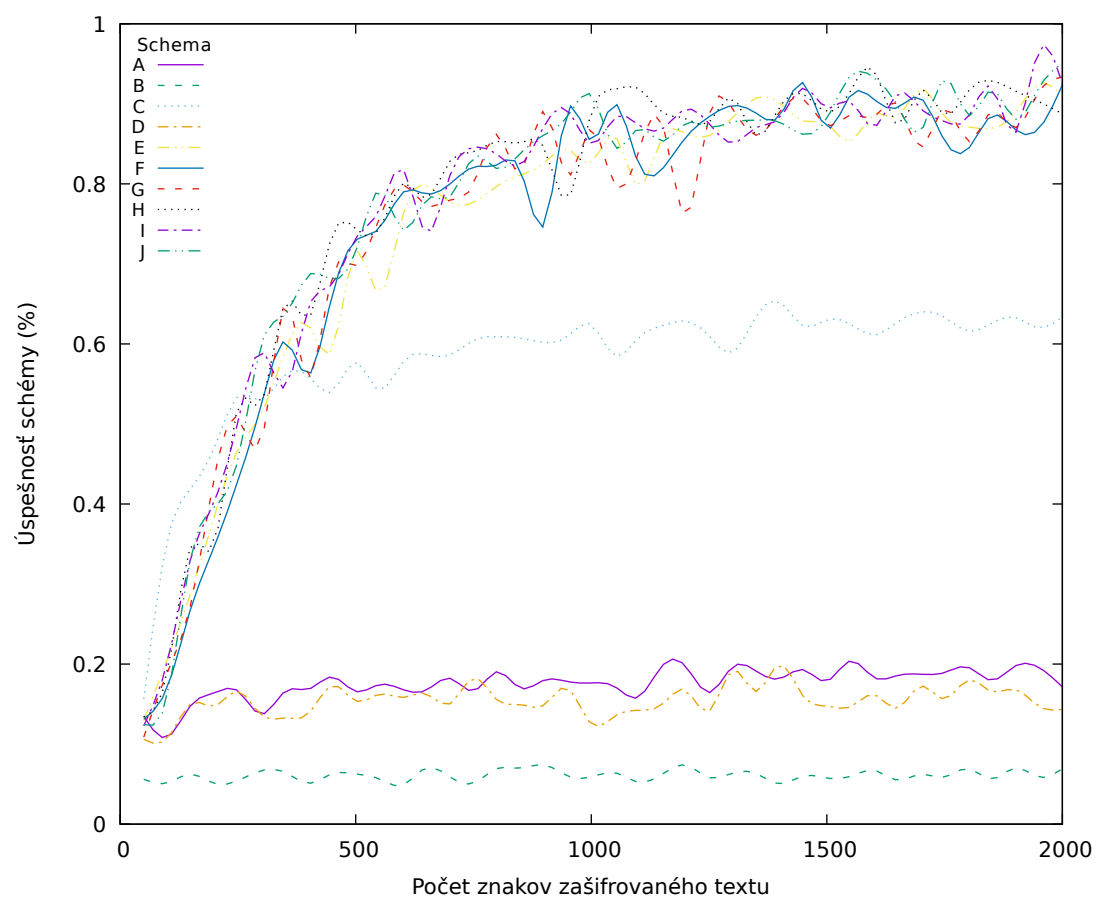
/bibliography.bib

- bibliography file

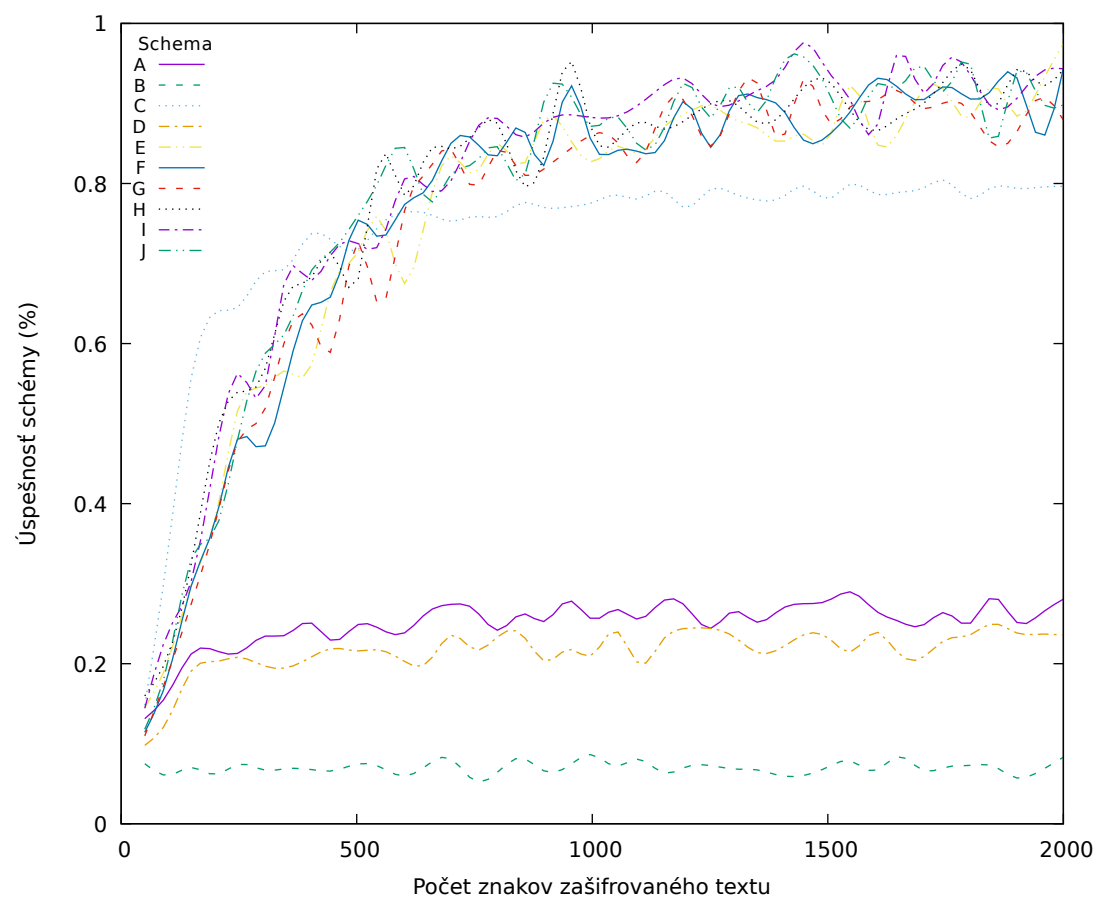
/attachmentA.tex

- this very file

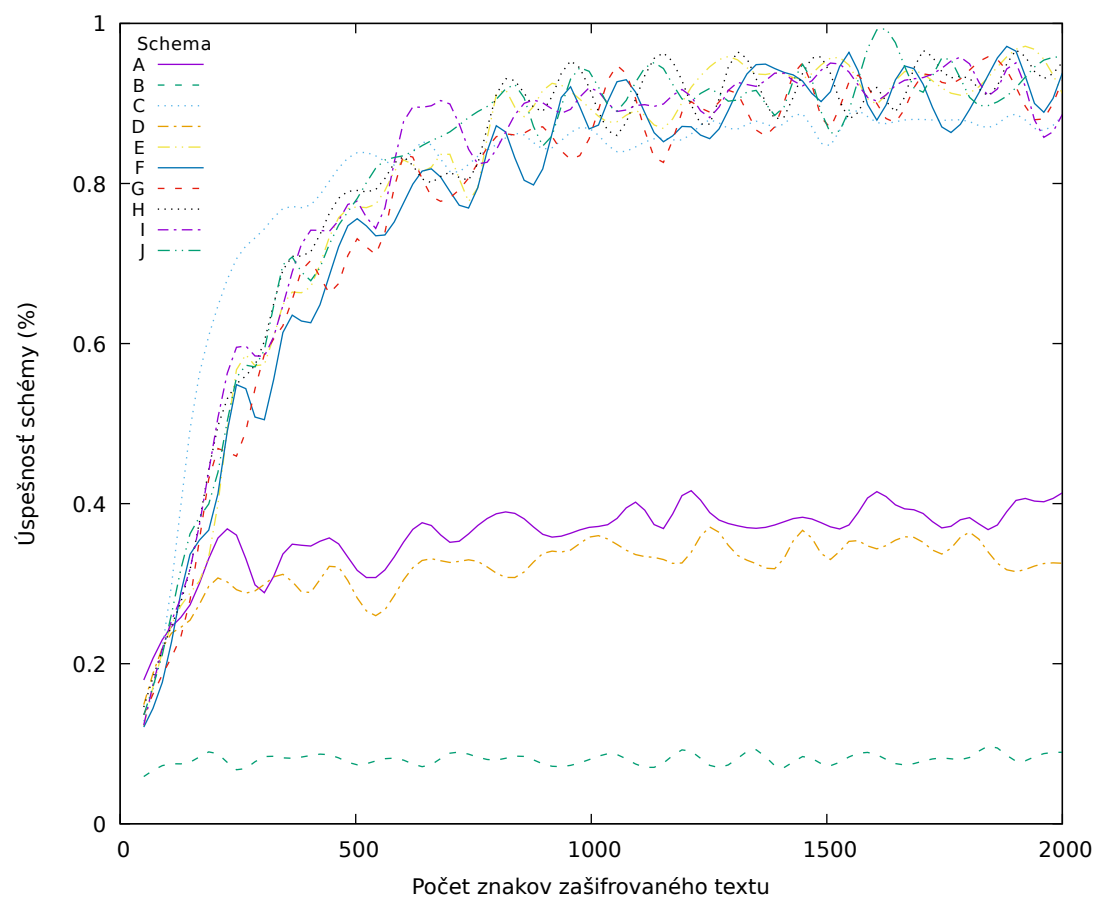
B Výsledky Genetického algoritmu



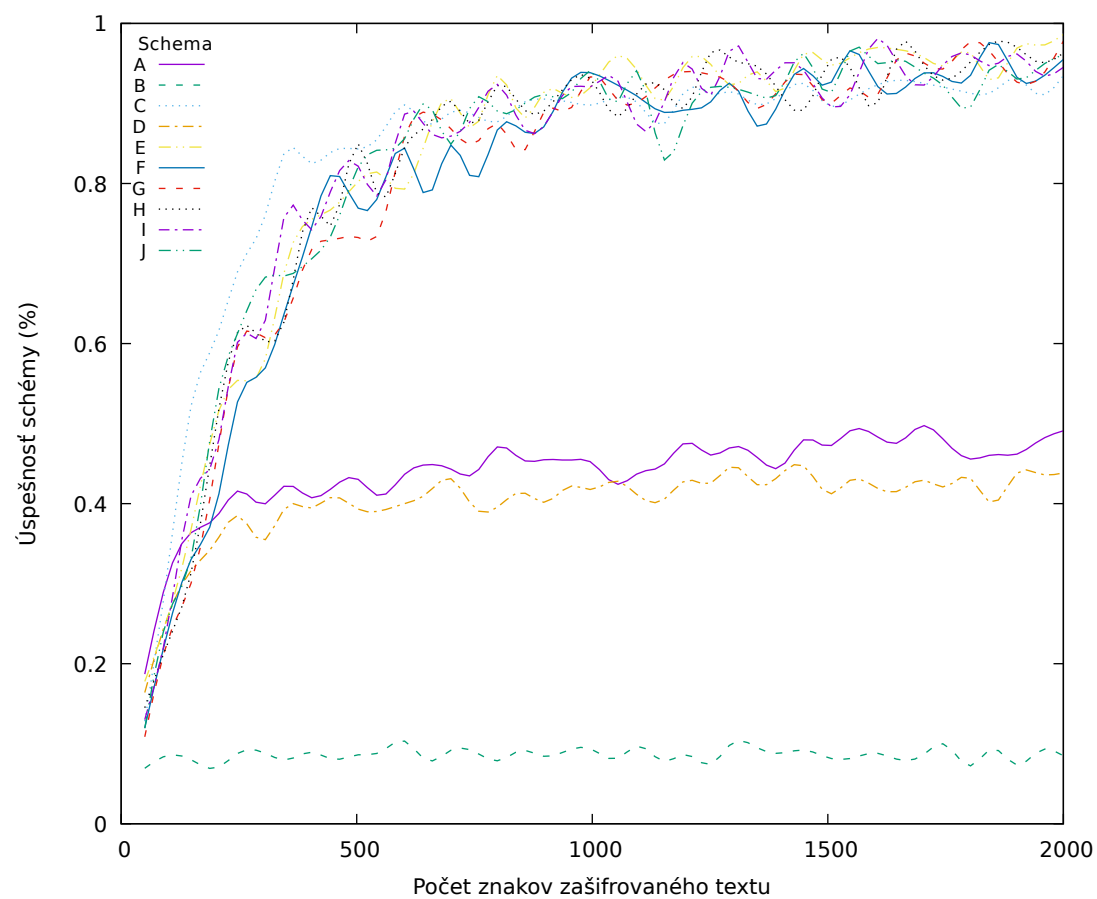
Obrázok B.1: Počet iterácií: 10000, počiatočná populácia: 10



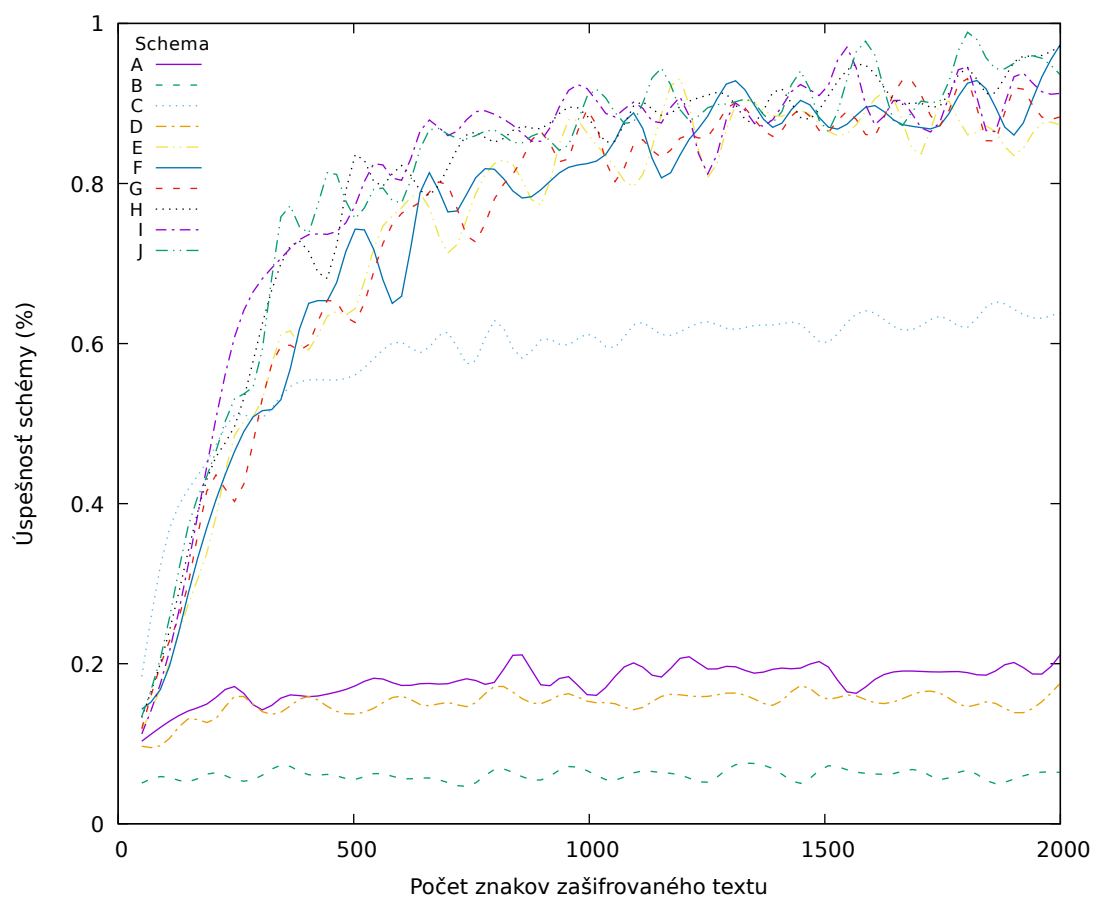
Obrázok B.2: Počet iterácií: 10000, počiatočná populácia: 20



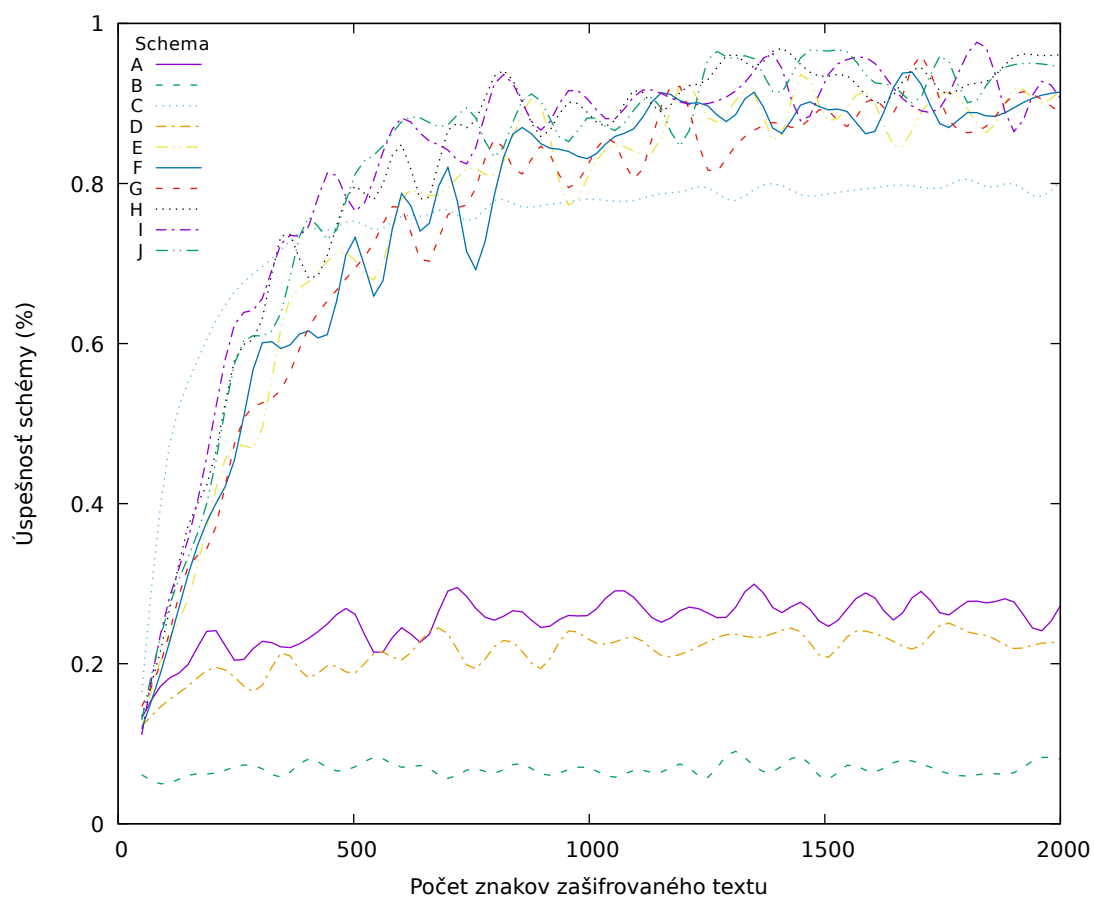
Obrázok B.3: Počet iterácií: 10000, počiatočná populácia: 50



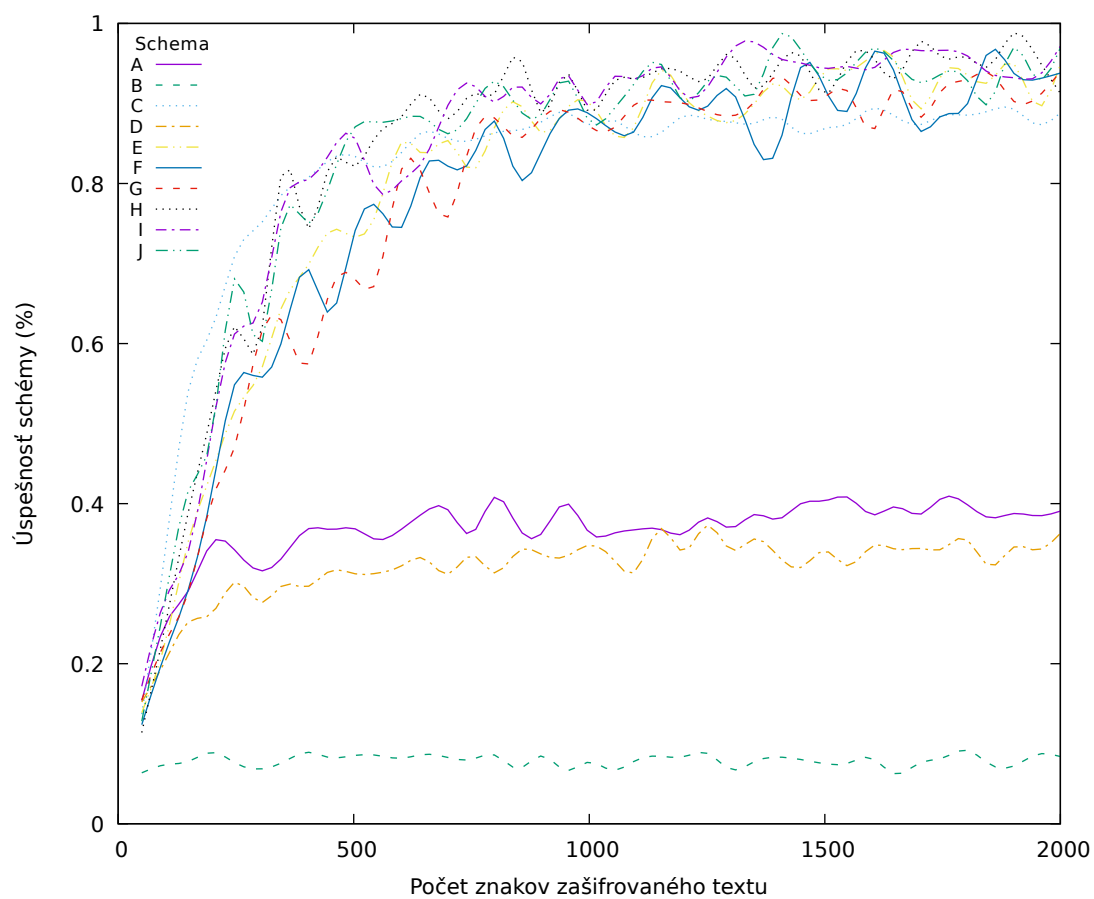
Obrázok B.4: Počet iterácií: 10000, počiatočná populácia: 100



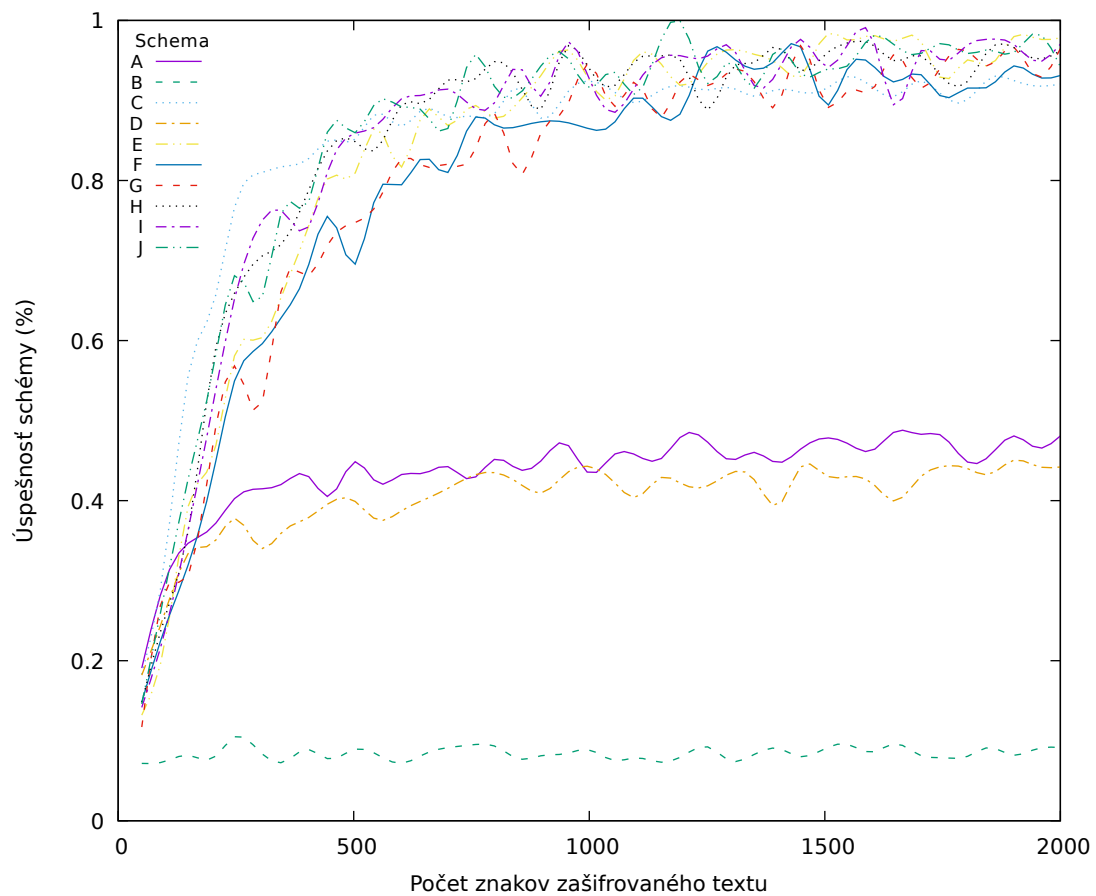
Obrázok B.5: Počet iterácií: 50000, počiatočná populácia: 10



Obrázok B.6: Počet iterácií: 50000, počiatočná populácia: 20



Obrázok B.7: Počet iterácií: 50000, počiatočná populácia: 50



Obrázok B.8: Počet iterácií: 50000, počiatočná populácia: 100