ZÁPADOČESKÁ
UNIVERZITA
V PLZNI

KIV/PPR
Semetrální práce 05
-
Hledání výskytů řetězce v souboru

Autor: Kateřina Blažková
Datum: 12.01.2006
Os. č.: A04040
Narozena: 16. 4. 1982
E-mail: kablazka@students.zcu.cz

Obsah

1. Zadání

05 - Realizujte hledání všech výskytů daného podřetězce x1 v textovém souboru. Výstupem programu je seznam dvojic (řádek, pozice).
- povinné parametry:
- řetězec x1
- název souboru

Povinným parametrem programu je počet procesů/vláken a velikost kusu práce přidělované procesům (dynamické přidělování práce (*Load-balance*)).

Úlohu individuálně vypracujte ve dvou verzích:

1. Paralelní program pro systém se sdílenou pamětí (použitelné prostředky: prog. jazyk Java nebo vlákna POSIX).
2. Paralelní program pro systém s distribuovanou pamětí (použitelné prostředky: PVM nebo MPI) .

2. Analýza úlohy

Úlohu lze řešit jako úlohu farmer-worker:
Farmer:

- obsahuje data
- rozděluje práci workerům

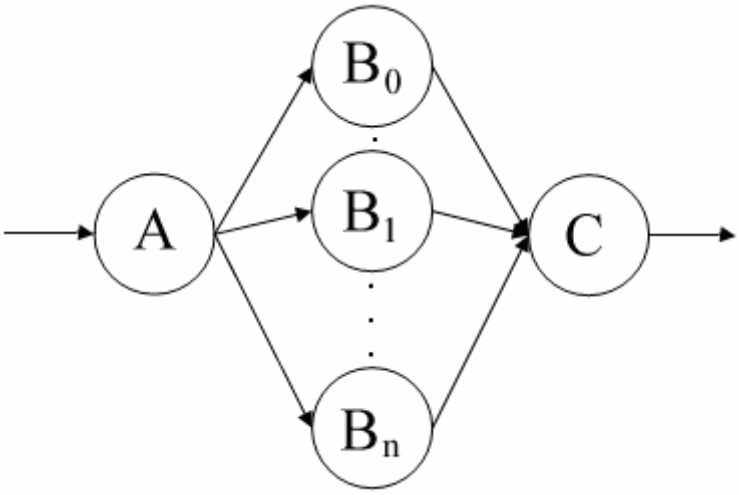
Workeři:

- vykonávají práci
- výsledek odesílají zpět farmerovi.

Postup zpracování lze rozdělit na následující kroky:

- A - načtení souboru do paměti
- B - nalezení výskytů hledaného řetězce
- C - zkompletování všech výskytů

Je tedy vhodné paralelizovat druhý krok. Proces můžeme vyjádřit následujícím precedenčním grafem.



Obr.1 – Precedenční graf vyjadřující postup zpracování

3. Implementace

Zadaná úloha byla řešena pro dva systémy:

- Paralelní program pro systém se sdílenou pamětí řešený pomocí programovacího jazyka **JAVA**.
- Paralelní program pro systém s distribuovanou pamětí řešený pomocí programovacího jazyka C a systému **MPI**.

4. Java

Paralelní procesy jsou vytvořena jako vlákna komunikující přes sdílenou paměť. Sdílenou paměť představuje třída s krytickou sekcí chráněnou monitorem.

4.1. Instalace

Program byl odladěn na java sun jdk-1.5. Pro spuštění programu z příkazové řádky je třeba mít nainstalované jdk a správně nastavené cesty v systémové proměnné PATH.

4.2. Popis ovládání

Aplikace pro Javu se nachází v adresáři PPR\PPR1.

Script pro Win: PPR\PPR1\run\run.bat
Script pro Linux: PPR\PPR1\run\run.sh

4.2.1. Spuštění na jumbo.fav.zcu.cz

- Nakopírovat adresář s obsahem run na jumbo.fav.zcu.cz
- Příkaz pro přiřazení práv: *chmod a+x run.sh*
- Příkaz pro spuštění: *run.sh*

Nebo

Program se spustí z příkazové řádky příkazem:
java –jar PPR1.jar <počet procesů> <velikost práce> <řetězec> <vstupní soubor>
např.
java -jar PPR1.jar 10 20 a data\small.txt
Parametry:

- *<počet procesů>* hodnota určující, kolik vláken bude program pro výpočet používat
- *<velikost práce>* počet řádek, kolik bude jedno vlákno zpracovávat
- *<řetězec>* řetězec, který se má vyhledávat
- *<vstupní soubor>* cesta a jméno souboru obsahující data

Pokud nebude zadán žádný parametr, aplikace vypíše informaci, kolik a jaké parametry mají být zadány.
Všechny vstupní parametry jsou náležitě ošetřeny, pokud uživatel zadá chybný počet parametrů nebo chybný parametr, je na to upozorněn chybovým hlášením a program je ukončen.

4.3. Vstupní data

Vstupní data jsou načítána ze souboru, který je zadán jako čtvrtý parametr. Počet a délka řádku není nijak omezena.

4.4. Výstup

4.4.1. Konzolový výstup

Do konzole je vypisována průběžná komunikace mezi vlákny a další důležité informace. Výskyty řetězce v sobouru jsou poté vypsány do konzole ve formátu:
line: <řádka> position: <pozice>

- *<řádka>* hodnota, na které řádce byl nalezen výsky, číslováno od 1
- *<pozice>* ukazje na pozici v řádce, kde byl výskyt nalezen, číslováno od 1

4.4.2. Výstup do souboru

Veškeré informace vypisující se do konzole jsou logovány do souboru *StringFinder.log.* Úroveň logování je možno nastavit v souboru *log4j.properties.*

4.5. Popis implementace

Podrobný popis jednotlivých metod lze nalézt v přiloženém *JavaDocu*. Zde proto uvedu pouze stručný popis fungování aplikace a tříd.

Program pracuje na principu farmer/worker s dynamickým přidělováním práce.

4.5.1. Bodové chování aplikace

- Třída StringFinder ošetří vstupní parametry, načte data ze souboru do paměti a vytvoří instanci třídy Farmer. Poté zavolá metodu Farmera na vrácení zpracovaných výsledků.

- Třída Farmer vytvoří n vláken (třídy Worker), které slouží jako Workeři pro zpracování kusu práce. Dále se stará o rozdělování kusů práce pro Workery a kompletování výsledků vrácených od Workerů.

- Každý Worker se zeptá Farmera, zda je práce, a pokud ano, vezme si práci. Přidělený kus práce(počet řádek) zpracuje a výsledky vráti zpět Farmerovi.

- Farmer přijímá výsledky, ukládá je do paměti a počítá si, kolik výsledků již dostal. Pokud zjistí, že všechna práce je již zpracována, nastaví patřičné proměnné tak, aby Worker, až se zeptá, zda je práce, dostal zápornou odpověd.

- Takto tedy Farmer získává jednotlivé části výsledků, které kompletuje dohromady.

- Když se vrátí výsledky od všech Workerů, ukončí se v metodě Farmera cyklus, který čekal, až budou všechny výsledky zpracovány, a tato metoda vrátí zpracovaný setříděný výsledek třídě StringFinder

- Třída StringFinder zpracované výsledky vypíše.

4.6. Stručný popis tříd

4.6.1. Třída StringFinder

Hlavní třída, vypíše info o programu, ošetří vstupní parametry, načte data ze souboru do paměti. Vytvoří instanci třídy Farmer, zavolá metodu Farmera, která vrací vzpracované výsledky.

4.6.2. Třída Farmer

Vytvoří Workery, nastartuje je. Obsahuje metodu ke zjištění, zda je práce, metodu pro rozdělení práce a metodu pro předání výsledků.

4.6.3. Třída Worker

V metodě run() se dotáže na práci, práci zpracuje a předá výsledky Farmerovi.

4.6.4. Třída PieceOfWork

Třída uchovává všechna data, potřebná pro zpracování Workerem a uložení výsledků.
Struktura

- List<String> data – list, obsahující všechny řádky načtené ze souboru
- String findString – hledaný řetězec
- int fromLine – řádek od kterého se mají data zpracovávat
- int toLine – řádek do kterého se mají data zpracovávat
- ArrayList<Occurence>() array – arrayList výsledků

4.6.5. Rozhraní IPieceOfWork

Rozhraní třídy PieceOfWork

4.6.6. Třída Occurence

Třída uchovává řádek a pozici jednoho výskytu řetězce.
Struktura

- int line – řádka, kde se výskyt nachází
- int position – pozice v řádce, kde se výskyt nachází

4.6.7. Rozhraní IOccurence

Rozhraní třídy IOccurence

4.6.8. Knihovna log4j-1.2.14.jar

Logování je nastaveno úroveň INFO, pro podrobnější výpisy je možno nastavit výpis TRACE v souboru *log4j.properties*, kde je možné také upravit formu výpisů dle potřeby. Podrobnější info viz. http://logging.apache.org/log4j/docs/download.html

Výpis se provádí do konzole a do souboru *StringFinder.log*.

5. MPI

MPI je programový prostředek pro realizaci masivně paralelních výpočtů v paralelním stroji s distribuovanou pamětí. Programuje se v C nebo ve Fortranu, přičemž se využívá interface knihovny MPI. Využití MPI do značné míry "odstíní" programátora od konkrétní konfigurace výpočetního prostředí (emuluje homogenní multiprocesorové výpočetní prostředí). MPI je zejména vhodné k programování paralelních algoritmů, které využívají regulární datové struktury (t.j. homogenní vektory a matice, viz klasická numerická matematika)[1].

5.1. Přehled vlastností a funkcí MPI

5.1.1. Základní funkce MPI

- `int MPI_Init (int* argc, char*** argv)`
  Inicializace MPI výpočtu, `argc` a `argv` jsou argumenty hlavního programu.
- `int MPI_Comm_size (MPI_Comm comm, int* adr_size)`
  Zjištění počtu procesů, počet se dosadí do proměnné odkazované parametrem `adr_size`. `MPI_Comm` je typ "komunikátor", pokud při volání dosadíme za parametr `comm` hodnotu `MPI_COMM_WORLD`, zjistíme počet všech vytvořených procesů aplikace N.
- `int MPI_Comm_rank (MPI_Comm comm, int* adr_rank)`
  Zjištění čísla procesu v rámci "komunikačního světa" `comm`. Čísla jsou v rozmezí 0 až M-1, kde M je "rozměr komunikačního světa" reprezentovaného komunikátorem comm (M = N, pokud dosadíme za comm hodnotu `MPI_COMM_WORLD`).
- `int MPI_Finalize (void)`
  Ukončení výpočtu v MPI, provádí každý proces.
- `int MPI_Send (void* adr_buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  Odeslání zprávy s typem tag v komunikačním světě `comm` procesu `dest`. Odesílá se zpráva z bufferu `buf` obsahující `count` položek typu datatype.
  Čili buffer pro zprávu je na rozdíl od PVM kdekoliv v datech programu (zadá se adresa příslušného pole). Za datatype se dosazují buď primitivní typy MPI, například `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`, nebo strukturované typy vytvořené z primitivních
- `int MPI_Recv (void* adr_buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *adr_status)`
  Blokující příjem zprávy. Parametry mají analogický význam jako u `MPI_Send`. Navíc je parametr `adr_status`, odkazující kam se má uložit status příjmu zprávy (výstupní parametr). Status obsahuje položky: `status.MPI_SOURCE` od koho zpráva přišla a `status.MPI_TAG` jakého typu zpráva přišla. Dosadí-li se za source hodnota `MPI_ANY_SOURCE` a za tag `MPI_ANY_TAG`, přijme se jakákoliv zpráva a z uvedených položek výstupního parametru status se dá zjistit co to vlastně přišlo.

5.1.2. Globální operace MPI

Globální operace jsou operace, do kterých jsou zapojeny všechny procesy patřící do téhož "komunikačního světa" (tj. jedním parametrem funkcí pro globální operace je příslušný komunikátor). Funkci globální operace volají všechny zúčastněné procesy (což je

pochopitelné, protože typicky procesy běží podle téhož programu), přičemž jejich činnost se v obecném případě liší podle čísla procesu. Globální operace jsou "ušité" na manipulace s regulárními datovými strukturami, konkrétně s homogenními vektory a maticemi.

Globální operace v PVM nejsou (kromě broadcastu) a musely by se pracně rozepsat do posloupnosti operací `send()` a `receive()`. Globální operace MPI lze rozdělit na tři skupiny - synchronizace, přesuny dat a redukční operace.

5.1.2.1. Synchronizace

Každý komunikátor mj. realizuje bariéru, na které se mohou všechny jeho procesy synchronizovat voláním funkce

```
int MPI_Barrier (MPI_Comm comm)
```

Volání této funkce je tedy blokující a výpočet každého procesu pokračuje následujícím příkazem až poté, kdy se na bariéře "sejdou" všechny procesy patřící do "komunikačního světa" `comm`.

5.1.2.2. Přesuny dat

Jedná se o operace s charakterem broadcastu, shromáždění či rozptýlení dat a tzv. redukční oprace.

- `int MPI_Bcast (void* adr_buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
  Operace broadcast. Má v zásadě stejné parametry jako `MPI_Send()`. Proces s číslem `root` vysílá, ostatní zprávu přijímají (čili root používá buffer jako vysílací, ostatní jako přijímací). Proti `send()` chybí tag - pro "globální" komunikační operaci nemá význam - všichni aktéři komunikace prochází tímtéž bodem programu a tudíž vědí, "o co při komunikaci jde".
- `int MPI_Gather (void* adr_inbuf, int incnt, MPI_Datatype intype, void* adr_outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)`
  Proces s číslem `root` shromažďuje data od všech procesů včetně sebe. Čili všechny procesy vysílají data (`count` položek typu `intype`) z bufferu `inbuf` a proces `root` je zapisuje do bufferu `outbuf` - samozřejme je zapisuje v pořadí podle čísel vysílajících procesů (tj. nikoliv tak, jak zprávy došly). Parametr `outbuf` (a též `outcnt` a `outtype`) využije jen proces `root`. Za `incnt` a `intype` se normálně dosadí stejné hodnoty jako za `outcnt` a `outtype`.
  Výstupní buffer zřejmě musí být M-krát větší než vstupní buffer, kde M je rozměr (počet procesů) komunikátoru comm.
- `int MPI_Scatter (void* adr_inbuf, int incnt, MPI_Datatype intype, void* adr_outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)`
  Inverzní operace k `Gather()`, tj. proces s číslem `root` "rozptyluje" data z bufferu `inbuf` všem ostatním procesům včetně sebe. Vstupní buffer musí obsahovat M částí dat (obecně různých, jedna část je pole `count` položek typu `intype`), přičemž i-tá část se pošle do bufferu `outbuf` procesu s číslem i. Parametr `inbuf` (a též `incnt` a `intype`) využije jen proces `root`. Za `incnt` a `intype` se normálně dosadí stejné hodnoty jako za `outcnt` a `outtype`.

Vstupní buffer zřejmě musí být M-krát větší než výstupní buffer, kde M je rozměr (počet procesů) komunikátoru comm.

5.1.2.3. Redukční operace

Redukční operace má M operandů umístěných ve vstupních bufferech komunikujících procesů. Výsledek operace se zapisuje do výstupního bufferu buď jednoho určeného procesu (root) nebo všech procesů.

- `int MPI_Reduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
  Za parametr op se dosazuje typ realizované operace, kde použitelné symbolické hodnoty typu jsou
  `MPI_MAX`, `MPI_MIN` (maximum, minimum),
  `MPI_SUM`, `MPI_PROD` (součet, součin),
  `MPI_LAND`, `MPI_LOR`, `MPI_LXOR` (logické operace)
  `MPI_BAND`, `MPI_BOR`, `MPI_BXOR` (logické operace se všemi bity)
  Operandy jsou ve vstupních bufferech procesů, výsledek je ve výstupním bufferu procesu `root`.
- `int MPI_Allreduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
  Funguje jako předchozí operace, ale výsledek dostanou do výstupního bufferu všechny zúčastněné procesy, tudíž chybí parametr root, který pak nemá smysl.

5.1.3. Komunikátory

- `MPI_Comm_rank()`
  vrací počet procesů komunikátoru, základní funkce MPI - viz dříve v části 2
- `MPI_Comm_dup()`
  vytváří duplikát komunikátoru
- `MPI_Comm_split()`
  "štěpí" kpomunikátor na dva jiné, čili rozděluje jednu skupinu procesů na dvě jiné
- `MPI_Comm_free()`
  uvolňuje (likviduje) komunikátor (samozřejmě nikoliv procesy, které komunikátor reprezentuje)
- `MPI_Intercomm_create()`
  vytváří tzv. "interkomunikátor" jakožto prostředek komunikace mezi dvěma skupinami procesů.

5.1.4. Datové typy pro komunikaci

- `MPI_Type_contiguous()`
  Tzv. "typový konstruktor", definuje "spojité" (tj. "bez mezer") pole prvků jiného (i neprimitivního) typu.
- `MPI_Type_vector()`
  Tzv. "typový konstruktor", definuje "pole bloků" prvků jiného (i neprimitivního) typu, mezi bloky může být "mezera".
- `MPI_Type_indexed()`
  Tzv. "typový konstruktor", definuje "nepravidelné pole bloků" prvků jiného (i

neprimitivního) typu, bloky mohou být různě dlouhé a mohou mezi nimi být různě dlouhé mezery.

- `MPI_Type_commit()`
  Voláním této funkce se typ (dynamicky vytvořený některou z předcházejících operací) "přihlásí do MPI", a pak se teprve může používat v komunikačních operacích.
- `MPI_Type_free()`
  Voláním této funkce se typ (vytvořený voláním některého z výše uvedených konstruktorů) dynamicky zruší a už se nesmí používat v komunikačních operacích.

5.2. Popis ovládání

Aplikace pro MPI se nachází v adresáři PPR\PPR2\mpi.

5.2.1. Spuštění na hydra.fav.zcu.cz

- Nakopírovat adresář mpi na hydra.fav.zcu.cz
- Příkaz pro přeložení: *make*
- Příkaz pro spuštění: *make test*

Nebo

Program se spouští z příkazové řádky příkazem:
mpirun -np <počet procesů>./StringFinder <řetězec><vstupní soubor>
např.:
mpirun -np 6 ./StringFinder a ./small.txt
Parametry:

- *<řetězec>* řetězec, který se má vyhledávat
- *<vstupní soubor>* cesta a jméno souboru obsahující data

Pro základní operace s programem je přiložen makefile, který umožňuje standardní operace:

- make – přeložení programu
- make clean – vymazání zkompilovaných částí programu
- make test – spuštění programu s testovací konfigurací

Pro vytvoření spustitelného souboru stačí v souboru se zdrojovými soubory spustit makefile příkazem “make”. Toto musí být spouštěno na stroji, kde je přístupné MPI prostředí.
Pro spuštění stačí v souboru se zdrojovými soubory napsat „make test”.

Přeložení:

- mpicc –o StringFinder StringFinder.c

Spuštění:

- mpirun -np 6 ./StringFinder *<řetězec> <vstupní soubor>*

Vymazání zkompilovaných částí:

- rm -f StringFinder *.o

5.3. Vstupní data

Vstupní data jsou načítána ze souboru, který je zadán v posledním parametru. Počet a délka řádku není nijak omezena. Soubor musí být ukončen znakem nové řádky(\n).

5.4. Výstup

Veškerá komunikace mezi Farmerem a Workery a ostatní důežité informace jsou vypisovány do konzole. Konečné výsledky jsou vypsány ve formátu:

FINAL RESULTS: string to find: 'a'

Thread No.: <číslo procesu> Line[<řádek>]= <pozice>;<pozice>;<pozice>;.....

- *<číslo procesu>* identifikační číslo procesu
- *<řádek>* čísla řádek číslovány od 1
- *<pozice>* čísla pozic číslovány od 1

5.5. Popis implementace

5.5.1. Bodové chování aplikace

Program pracuje následujícím způsobem:

- Root neboli Farmer provede načtení dat ze souboru, Worker procesy čekají než načtení skončí.
- Po načtení root zjistí, kolik má k dispozici procesů.
- Root nejprve rozešle všem Workerům data, potřebná ke zpracování - tedy 1 řádek, hledaný řetězec, číslo řádku.
- Worker data zpracuje, výsledky odešle Farmerovi, který je přidá do celkových výsledků práce.
- V cyklu se přijímají výsledky od Workera a posílají se data právě tomu stejnému Workerovi, dokud nějaká práce je(dokud se nerozeslaly všechny řádky souboru).
- Když je všechna práce rozeslána, Root počká na výsledky od všech procesů a jakmile zjistí že je všechna práce hotova, pošle info všem Workerům, aby se ukončily.
- Root zobrazí výsledky.

5.6. Popis funkcí a dat

5.6.1. Třída StringFinder.c

5.6.1.1. Struktura dat a výsledku:

```
typedef struct pieceOfWork {
  char *findString;
  int fromLine;
  int toLine;
  char **data;
  char **results;
} PIECEOFWORK;
```

5.6.1.2. int main (int argc, char *argv[])

Funkce zkontroluje počet parametrů a pokud je správný, inicializuje MPI
Pokud se jedná o proces root, volá funkci Farmer, pokud ne, volá funkci Worker.

5.6.1.3. int readFile(char* fileName)

Funkce pro načtení souboru do struktury, vrací 1 při úspěchu.

5.6.1.4. void farmer(char* stringToFind)

Alokuje místo pro celkové výsledky. Nejprve rozešle data všem Workerům a poté v cyklu přijímá zpracované výsledky pomocí funkce *MPI_Recv()* - (přijme délku výsledku, přijme výsledek(pole charů)),
a vysílá data tomu, od koho je přijal, pomocí funkce *MPI_Send()* - (pošle délku řádky, řádku, délku hledaného řetězce, hledaný řetězec a číslo řádky) dokud nerozešle všechnu práci (všechny řádky).

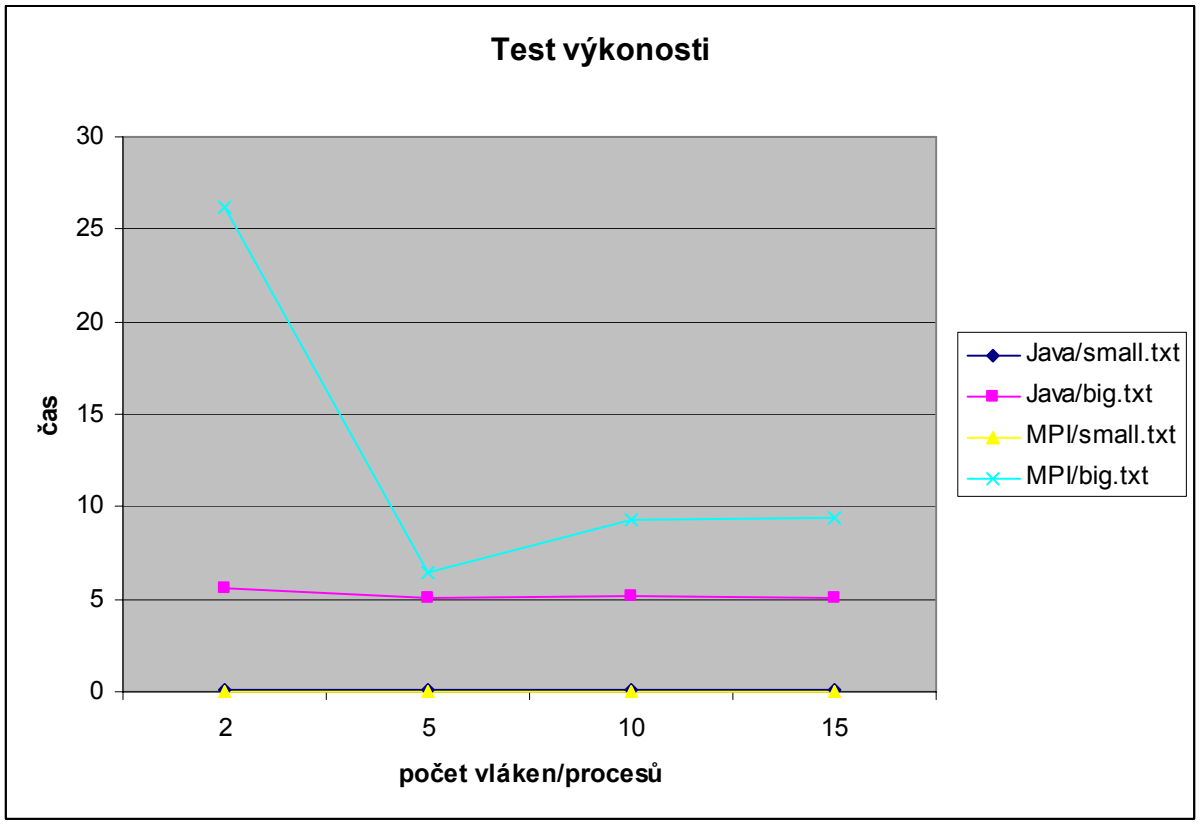
Farmer odesílá jednotlivé kusy práce(řádky) postupně v cyklu, jak mu přicházejí výsledky od jednotlivých Workerů. Pokud by tedy byl některý Worker více vytížený(„pomalejší“), nemusí se v průběhu vždy čekat na jeho doběh, ale práce bude rozdělována rychlejším Workerům.

5.6.1.5. void worker()

Přijímá data pomocí funkce *MPI_Recv() – (*přijme délku řádky, řádku, délku hledaného řetězce, hledaný řetězec a číslo řádky*),*
prohledá data a výsledky hledání odesílá procesu root pomocí funkce MPI_Send() – (odešle délku výsledku, odešle výsledek(pole charů)).

6. Test rychlosti zpracování

Počet vláken/procesů	2	5	10	15
Java/small.txt	0,078	0,08	0,083	0,086
Java/big.txt	5,578	5,079	5,125	5,109
MPI/small.txt	0,0092	0,0048	0,004	0,0034
MPI/big.txt	26,19	6,47	9,32	9,43



Z grafu je vidět, že paralelní výpočet realizovaný v Javě testovaný na single core processoru vykazuje relativně konstantní výsledky, avšak týž paralelní výpočet realizovaný v MPI na serveru hydra.fav.zcu.cz dosahuje vykazuje při běhu s 5ti a více procesy zajímavého urychlení.

7. Závěr

7.1. Java

Program pracuje dle zadání. Ráda jsem si oživila práci s vlákny. Programování v jazyce Java pro mě bylo mnohem více příjemnějším a zábavnějším než v jazyce C.

7.2. MPI

Nejprve jsem chtěla program napsat v PVM, ale když jsem nenalezla způsob, jakým vynutit, aby se výpisy *printf* na straně workerů vypisovaly do konzole, jsem další programování v tomto prostředí zavrhla. Bez průběžných výpisů by bylo program obtížné odladit. Program jsem tedy poté napsala v MPI, ve kterém funguje dle zadání.

7.3. Shrnutí

Tato semestrální mě opět přesvědčila o tom, že programování v jazyce C mi vyhovuje mnohem méně než programování v jazycích běžících nad VM.

Podle mého názoru by mnohým studentům pomohlo, kdyby prostředí MPI či PVM bylo na hydra.fav.zcu.cz zprovozněno také pro jazyk Java.

Literatura

[1] K.Ježek, P.Matějovic, S.Racek: Paralelní architektury a programy skripta ZČU, FAV, Plzeň 1999

[2] http://www.kiv.zcu.cz/~stracek/ppr/

[3] https://jumbo.fav.zcu.cz/vyuka/

[4] Herout, P.,: Učebnice jakyka C, Kopp, České Budějovice, 2001

[5] Herout, P.,: Učebnice jakyka C - 2.díl, Kopp, České Budějovice, 2001

[6] http://logging.apache.org/log4j/docs/download.html