Západočeská univerzita v Plzni
Fakulta aplikovaných věd
katedra informatiky a výpočetní techniky

Semestrální práce z předmětu KIV / PPR

Příklad 8 - Výpočet hmotnosti kvádru

Jméno: Jiří Frolík
Osobní číslo: A06081
Obor: ININ - SWI
Datum narození: 20. 3. 1984
Datum odevzdání: 15. 1. 2008
E-mail: j-frolik@seznam.cz

Obsah

1 Zadání

Úkolem práce je vytvořit:

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

Zadání číslo 8

Realizujte výpočet celkové hmotnosti zadaného kvádru prostoru (0,0,0,x,y,z). Je dána funkce hustoty prostoru ρ(x,y,z).

Povinné parametry

- počet vláken
- velikost kusu práce přidělovaného vláknům
- definice kvádru (x,y,z) (jeden roh kvádru se nachází v počátku souřadného systému)
- integrační krok dx, [dy, dz] - může být stejný pro všechny osy
- nějaký způsob vybírání funkcí (ne překompilováním programu)

Povinné funkce

ρ (x,y,z) = 1
ρ (x,y,z) = x
ρ (x,y,z) = x + y + z

Vzorce pro výpočet hmotnosti

(**Vz 1**) $M = \sum dM$ → celková hmotnost - součet dílčích výsledků

(**Vz 2**) $dM(x, y, z) = dx \cdot dy \cdot dz \cdot \rho\left(x + \frac{dx}{2}, y + \frac{dy}{2}, z + \frac{dz}{2}\right)$ → hmotnost kvádříku (dx,dy,dz)

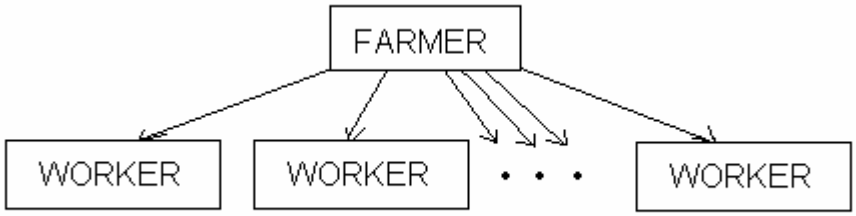
2 Teorie

2.1 Analýza problému a dekompozice úlohy

Úkolem práce je vytvořit paralelní program, který by provedl výpočet hmotnosti zadaného kvádru podle výše uvedeného vzorce. Ať už tedy budu využívat vláken v Javě nebo prostředku pvm, musím nejdříve zadanou úlohu rozložit na menší části. V pokynech ke zpracování semestrální práce je též uvedeno,že data by měla být dělena mezi procesy dynamicky (load-balance - algoritmus vyvažování zátěže - a model farmer-worker).

2.1.1 Farmer - worker

Jedná se o plánování úloh typu task-farming, kdy je vždy jeden řídící proces (farmer), který uchovává a rozděluje data ke zpracování do malých částí. Ty posílá ke zpracování ostatním uzlům (worker), které po provedení výpočtu své části práce pošlou výsledky zpět farmerovi. Ten dílčí výsledky zpracovává a sestavuje z nich celkový výsledek výpočtu.



Obr. 1 - Model farmer - worker

2.1.2 Dekompozice úlohy

V případě Javy budou funkce jak farmera, tak workerů obstarávat vlákna Thread. Bylo by tedy vhodné, aby byl každý typ uzlu představován jednou třídou. V případě pvm a programovacího jazyka C budou toto rozdělení zajišťovat samostatné moduly.

Dále budu v každém případě potřebovat nějakou strukturu, která by mi uchovávala načtená data potřebná k výpočtu a zadané parametry.

Kromě poměrně jasných částí programu uvedených níže (farmer, worker, kvádr) je ještě nutná část, kde jsou uchovány **parametry výpočtu**, tedy hodnoty integračních kroků **dx**, **dy**, **dz**, dále pak **počet workerů** účastnících se výpočtu, **množství přidělené práce** (postupně posíláno workerům) a také **typ funkce** ρ(x,y,z), podle které se vypočte hustota v daném bodě kvádru.

V rámci dobré přehlednosti a modifikovatelnosti, co se týče různých typů funkcí ρ(x,y,z) pro výpočet hustoty, bude dobré oddělit implementace těchto funkcí také do samostatné třídy / modulu.

Program (ať už v Javě nebo C) je strukturován následovně:

Farmer

Eviduje množinu workerů, kteří se účastní výpočtu. Provádí rozdělení kvádru na malé části (viz kap. 2.2.1). Tyto části pak zasílá jednotlivým workerům ke zpracování. Dílčí výsledky výpočtu opět od workerů sbírá a sestavuje je do celkového výsledku výpočtu, který představuje spočtenou hmotnost zadaného kvádru.

Worker

Zahrnuje vzorec (**Vz 2**) pro výpočet hmotnosti části kvádru. Potřebné parametry výpočtu přebírá od farmera, provede svůj výpočet tak, že podle zadaných parametrů dx, dy, dz a zadané funkce ρ(x,y,z) určuje hmotnost částí přiděleného úseku kvádru. Pak dá farmerovi k dispozici výsledek.

Kvádr

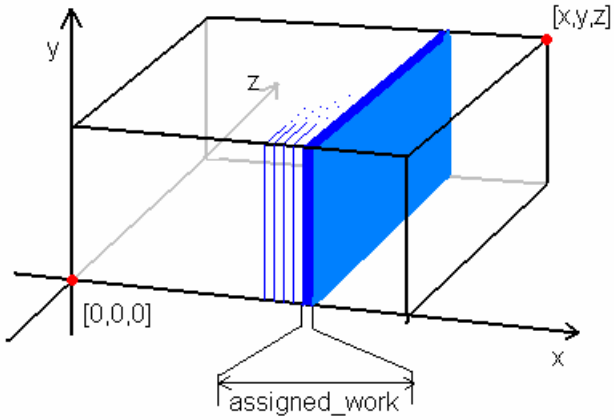
Struktura (ať už řešena třídou nebo strukturou v jazyce C) by měla obsahovat hodnoty, které jednoznačně vymezují kvádr v prostoru. Protože je v zadání uvedeno, že jeden vrchol kvádru se nachází v počátku souřadnic (a zřejmě budou hrany kvádru rovnoběžné s osami *x*, *y* a *z*), stačí uchovávat pouze souřadnice jednoho vrcholu navíc, a to vrcholu protilehlého počátku souřadnic po tělesové úhlopříčce (viz Obr. 2).

2.2 Algoritmus rozdělení kvádru a přidělení práce

2.2.1 Rozdělení kvádru

Aby mohl farmer nějak rozdělit práci mezi workery, musí existovat algoritmus, podle kterého by se dal rozdělit zadaný kvádr na malé kousky.

Protože existuje pevný parametr výpočtu „množství přidělené práce jednotlivým workerům“ (par. **assigned_work**), znamená to, že je nutné rozdělit kvádr na části právě tak velké, jako je zadaný parametr. Parametr udává **poměr velikosti** dané části ku velikosti celého kvádru. Bylo by složité rozdělovat kvádr po více osách najednou a dostat tak vždy malý kvádřík (přirovnám k silné cihlové zdi), je mnohem jednodušší zadaného parametru využít a kvádr rozdělit pouze podle jedné osy (rozhodl jsem se pro osu *x*). Vzniknou tak tenké, ale dlouhé a vysoké kvádry (přirovnám k rozdělení knihy na listy). Když nastavím daný parametr tak, aby číslo 1.0 tvořilo celý kvádr, tedy 100%, dostanu pak šířku jednoho „listu“ jednoduchým vynásobením parametru (například 0.02, což znamená, že takových „listů“ bude 50) a rozměru kvádru v ose x.



Obr.2 - Rozdělení kvádru podle osy x do částí o poměrné velikosti assigned_work

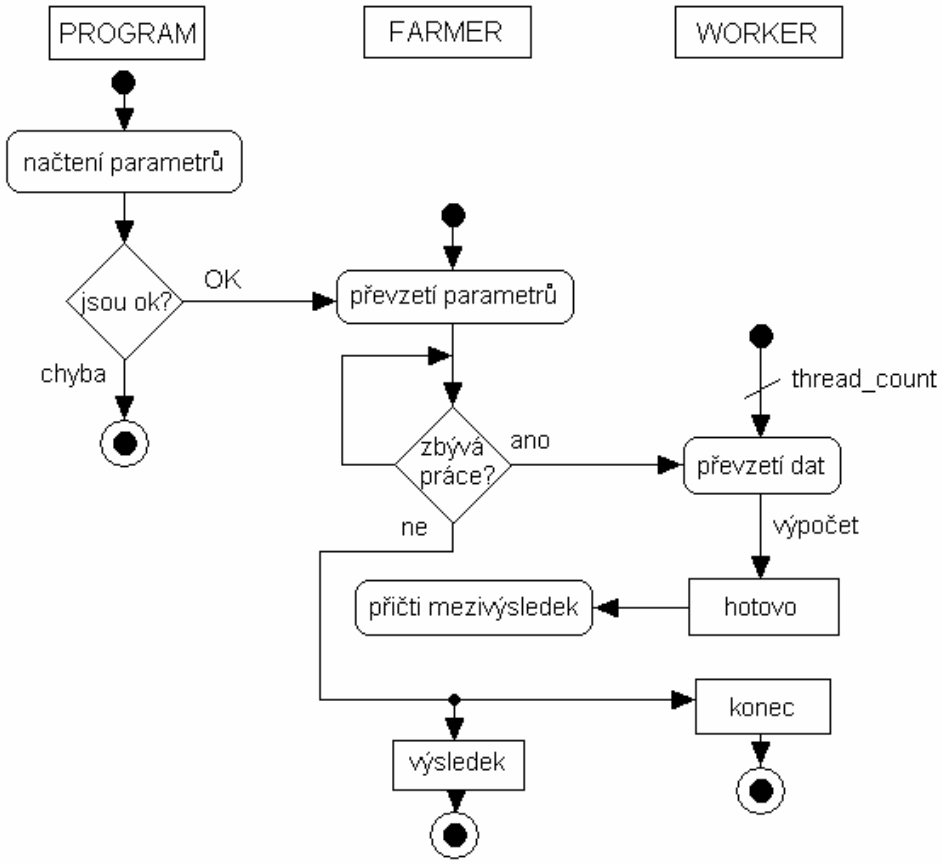
2.2.2 Přidělení práce

Podle výše uvedeného algoritmu je snadné rozdělit kvádr do menších částí. V cyklu farmera probíhá přidělování práce workerům v cyklu tímto způsobem: pokud je stále co počítat (množství přidělené práce je menší než 1.0), pošlu danému workerovi část kvádru k určení její hmotnosti. Pokud je to poslední část kvádru, bude množství přidělené práce větší nebo rovno hodnotě 1.0 a tehdy se rozesílání ukončí. Zde se může stát, že poslední přidělená

část kvádru (podle zadaného parametru) nemusí vyjít přesně do hodnoty 1.0. Výsledek výpočtu si farmer vždy převezme a připočte ho k hmotnosti už vypočtené části kvádru.

Pokud už není práce pro workery, farmer je jednoho po druhém ukončí.

2.3 Stručný diagram běhu programu



Obr. 3 - Stručný diagram běhu programu

3 Programátorská dokumentace

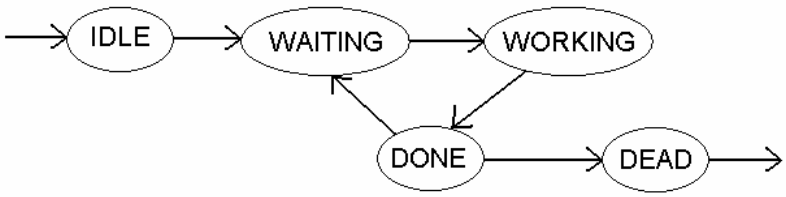
3.1 Implementace programu v Javě

Program jsem psal a testoval v prostředí Eclipse 3.1 s verzí javy 1.5.0_02-b09 pod MS Windows XP Home. K implementaci modelu farmer - worker využívá vláken (Thread). Skládá se ze 7 tříd.

3.1.1 Popis tříd

- **Kvadr.java** - třída uchovávající strukturu kvádru - obsahuje souřadnice dvou bodů, protilehlých po tělesové úhlopříčce. Jeden z nich leží defaultně v počátku souřadnic (x0,y0,z0), druhý bod (viz Obr. 2) je načítán jako parametr programu (x,y,z).

- **Zpracovani.java** - uchovává parametry výpočtu, jak byly načteny buď z příkazové řádky nebo z konfiguračního souboru. Obsahuje metodu checkAttributes(), která provádí kontrolu správnosti načtených parametrů. Například při záporných číslech, nebo když je zadáno desetinné číslo místo celého atd., je vrácena chyba a celý program se poté ukončí.
- **Farmer.java** - vlákno, které provádí rozdělení výpočtu na malé části a ty přiděluje jednotlivým workerům, na které si udržuje reference. Výsledky sestavuje do konečné hmotnosti. Obsahuje metodu assignToWorkerAt(int), která vypočte novou část kvádru a předá ji workerovi s daným číslem. Pak workerovi dá příkaz, aby začal pracovat (nastaví mu stav na WORKING). Když je worker DONE, farmer odebere výsledek a přidělí další práci. Farmer v cyklu takto zadává práci a shromažďuje výsledky. Podrobnější popis činnosti workera viz kap. 2.1.2 a odstavec Worker.java. Třída Farmer ještě provádí měření doby času výpočtu. Výsledky tohoto měření viz kap. 5.
- **Funkce.java** - obsahuje statické metody výpočtu hustoty v daném bodě. (podle zadání práce jsou tyto funkce povinné)
- **Worker.java** - vlákno, které provádí částečné výpočty hmotnosti kvádru. Parametry výpočtu jsou přidělovány farmerem (Farmer.java). Worker se nachází vždy v jednom z 5 stavů (IDLE - vytvořen, WAITING - čeká na data, WORKING - probíhá výpočet, DONE - výpočet hotov, výsledek lze převzít, DEAD - worker ukončen). Graf stavů workera viz Obr. 4. Farmer tyto hodnoty stavů přiděluje a testuje momentální stav workera. Výpočet probíhá podle zadaného vzorce, využívá statických metod třídy Funkce.java.
- **Hlavni.java** - hlavní třída programu. Načte požadované parametry výpočtu hmotnosti kvádru (buď ze souboru nebo z příkazové řádky) a spustí proces farmera.
- **Logger.java** - zajišťuje logování do souboru. Viz kap. 4.4.



Obr.4 - Stavy, ve kterých se nachází worker

3.2 Implementace programu v pvm (C)

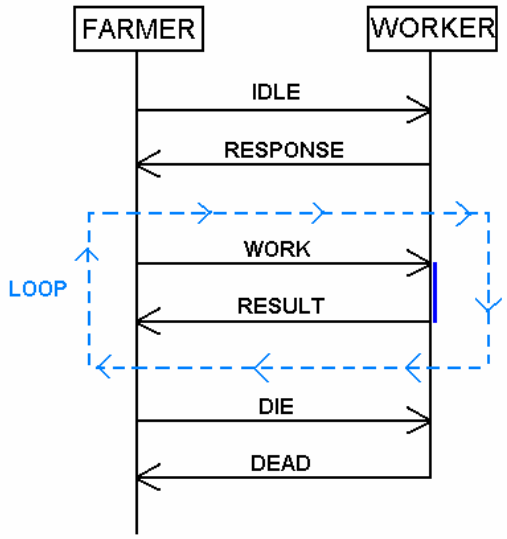
Program jsem psal v jazyce C a editoru PSPad 4.3.2 pod MS Windows XP Home. Přeložen a testován byl na stroji hydra.fav.zcu.cz.

Implementace modelu farmer - worker využívá prostředí pvm (funkce pvm_mytid(), pvm_spawn(), pvm_initsend(), pvm_pkint(), pvm_upkint(), pvm_mcast(), pvm_recv(), pvm_send() a pvm_exit(). Program zahrnuje celkem 4 *.c soubory a 3 hlavičkové *.h soubory. Jména i funkčnosti C funkcí odpovídají názvům metod v javovském programu. (Například sendToWorkerAt() odpovídá send_to_worker_at().)

3.2.1 Popis modulů

- **kvadr_zpracovani.h** - hlavičkový soubor obsahuje mimo několika konstant definici struktury KVADR (obdoba třídy Kvadr.java).

- **kvadr_farmer.c** - modul farmera, který převezme načtené parametry výpočtu, vytvoří požadovaný počet workerů (spuštěním pvm_spawn()) a postupně jim přiděluje dílčí úlohy. Princip je úplně stejný jako v javovském Farmer.java. Místo stavů workera si s ním vyměňuje zprávy přes pvm_send() a pvm_recv(). Přiděluje práci zprávou WORK a odebere výsledek při přijetí zprávy RESULT. Jakým způsobem se zprávy vyměňují, znázorňuje Obr. 5. Tento modul kromě zmíněné funkce ještě provádí měření času výpočtu. Výsledky tohoto měření viz kap. 5. Taktéž zajišťuje logování do souboru (viz kap. 6).
- **kvadr_funkce.h** - hlavičkový soubor k modulu kvadr_funkce.c
- **kvadr_funkce.c** - odpovídá souboru Funkce.java - obsahuje tři funkce ρ(x,y,z) k výpočtu hustoty v daném bodě kvádru.
- **kvadr_worker.h** - hlavičkový soubor k modulu kvadr_worker.c. Obsahuje definice číselných kódů svých stavů zpráv, které si vyměňuje s farmerem.
- **kvadr_worker.c** - modul workera, který přebírá parametry výpočtu od farmera a počítá dílčí hmotnost. Způsob výměny zpráv s farmerem viz Obr. 5.
- **kvadr_hlavni.c** - načte požadované parametry výpočtu hmotnosti kvádru (buď ze souboru nebo z příkazové řádky) a spustí farmera (příkazy init_farmer() a run_farmer()).



Obr.5 - Komunikace mezi farmerem a workerem v pvm

4 Uživatelská dokumentace

4.1 Přeložení programu - Java

Program lze přeložit buď dávkovým souborem `compile.bat` (případně pro linux `compile-linux.bat`) v adresáři `src`, nebo v tomtéž adresáři provést příkaz „`javac *.java`“. Po přeložení tímto příkazem je ale nutné nakopírovat vzniklé *.class soubory do adresáře `bin`, aby bylo možné program z onoho adresáře spustit souborem `run.bat` (viz 4.3). Pokud program přeložíte prvním způsobem, soubor `compile.bat` sám provede překlad a kopírování *.class souborů.

4.2 Přeložení programu - pvm

Nejprve se musíme ujistit, že se v kořenovém adresáři programu kromě všech potřebných *.c a *.h souborů vyskytuje i přiložený soubor `makefile`. Překlad programu pak provedeme příkazem „`make`“. Vytvoří se spustitelné soubory `farmer.exe` a `worker.exe`.

4.3 Spuštění programu

Ke spuštění programu v Javě slouží dávkový soubor `run.bat` v adresáři `bin` nebo příkaz „`java Hlavni`“. Oba způsoby se doplní požadovanými parametry výpočtu (viz 4.3.1).

Pvm program se spouští příkazem `farmer.exe` s parametry výpočtu (viz 4.3.1). Soubor worker.exe samostatně nespouštíme, neboť ten je spouštěn za běhu programu v rámci vytváření dělníků (workerů). Je potřeba mít spuštěného PVM daemona (pvmd) a nahraný soubor `hostfile` na server.

4.3.1 Parametry programu

Oba programy, jak v Javě, tak v pvm, se spouští se stejnými parametry na příkazové řádce v těchto tvarech:

- **bez parametru** - program vypíše nápovědu k jeho spuštění a skončí;
- **jeden parametr** - musí být jméno konfiguračního souboru, nalézajícího se v adresáři `config`; požadovaná struktura souboru viz (4.3.2);
- **všechny parametry výpočtu (7 nebo 9)**:

```
<x> <y> <z> <dx> [<dy> <dz>] <pocet_workeru> <pridelena_cast> <typ_func_ro>
```

Legenda:

x, **y**, **z** v desetinném tvaru jsou souřadnice „druhého rohu“ kvádru (viz Obr. 2);

dx, **dy**, **dz** - desetinná čísla vyjadřující přírůstky (integrační kroky) po jednotlivých osách;

pocet_workeru - celé číslo - počet vláken / procesů, které provádí dílčí výpočty hmotnosti kvádru podle přidělených dat;

pridelena_cast - desetinné číslo - část výpočtu (daná poměrem, kdy hodnota 1.0 = 100%), která je přidělována workerům;

typ_func_ro - celé číslo z množiny {0,1,2} označuje jednu z funkcí **ρ** (viz 1). Hodnota **0** vybere funkci vracející konstantní hustotu, **1** označuje funkci vracející x a **2** vybere funkce, která vrací součet (x+y+z);

Parametry **dy** a **dz** lze vypustit, pokud se rovnají **dx**; zadá se tedy 7 parametrů. Například:

```
farmer.exe 1 2 3 0.01 5 0.05 1
```

Při různých hodnotách dx, dy, dz je **nutné** uvést všech 9 parametrů:

```
farmer.exe 1 2 3 0.01 0.02 0.03 5 0.05 1
```

Parametry jsou na vstupu ošetřeny kvůli možné chybě při běhu programu. Pokud je program spuštěn s nesprávným počtem parametrů, nebo je zadáno záporné číslo, místo celého desetinné, neexistuje zadaný konfigurační soubor atd., je vypsána chybová hláška a program skončí.

4.3.2 Vlastnosti konfiguračního souboru

Konfigurační soubor libovolného jména musí být umístěn v adresáři `config`, který se v případě **Javy** musí nalézat na stejné úrovni adresářového stromu jako adresář `bin`. Naopak v případě **pvm** musí být adresář `config` přítomen v kořenovém adresáři programu se spustitelnými soubory farmer.exe a worker.exe.

Na rozdíl od zadávání parametrů příkazovou řádkou, zde je nutné, aby byly do souboru zadány **všechny** hodnoty a aby každý údaj byl na nové řádce. Postupně po sobě následují parametry kvádru, hodnoty přírůstku po jednotlivých osách a další parametry výpočtu:

Požadovaná struktura:	Například tedy:	Nebo:
kvadr.x	1.0	5
kvadr.y	2.5	12
kvadr.z	5.0	0.7
dx	0.01	0.005
dy	0.01	0.01
dz	0.01	0.05
pocet_workeru	5	12
pridelena_cast	0.05	0.1
typ_func_ro	1	0
<EOF>		

4.4 Logování

Jak program v Javě, tak v pvm, umožňuje logování průběhu výpočtu do souboru. Soubor se vždy jmenuje `log` a vytváří se v adresáři `log`, který musí být umístěn na stejné úrovni v adresářovém stromu, jako adresář `config` (viz 4.3.2). *Pozor* - v javovském programu se připisuje na konec stávajícího souboru, kdežto v pvm programu je vytvářen vždy soubor nový!

5 Měření a statistiky času výpočtu

5.1 Měření

Protože mne zajímalo, jak závisí doba výpočtu a přesnost výsledku na počtu workerů, množství přidělené práce jednomu workerovi a velikosti integračního kroku dx, provedl jsem několik měření. Měřil jsem jak výsledky javovského programu, tak programu pvm na clusteru

hydra.fav.zcu.cz. Každé měření (s výjimkou pravého sloupce) jsem prováděl nejméně 3x, do tabulky jsem zaznamenával prostřední hodnotu (abych eliminoval časové výkyvy, které nastávaly, někdy dost výrazné). Poznámka: hodnoty **d** a **aw** nejsou v tabulce lineárně rozloženy, není tedy možno vysledovat nějakou závislost na měnících se hodnotách **d** a **aw** přímo z sousedních sloupců tabulky.

Legenda k tabulkám:

N = počet workerů
t = čas výpočtu v ms
m = vypočítaná hmotnost
d = dx = dy = dz
aw = část přidělené práce jednomu workerovi (1.0 = 100%)

Program byl spouštěn s parametry: **`1.0 2.5 5.0 d N aw 0`**

Teoretická hodnota výsledné hmotnosti = **12,5**

Tab.1 - tabulka pro Javu - měřeno na Athlon XP2000+ 512RAM, WIN XP Home, verze jdk: 1.5.0_02-b09

N \ d / aw		0,05 / 0,1	0,01 / 0,1	0,01 / 0,05	0,005 / 0,01	0,001 / 0,01
2	**t**	**1662,0**	**1600,0**	**2868,0**	**14579,0**	**883189,0**
	m	14,165250	12,826602	12,700851	12,725588	12.505000
5	**t**	**1224,0**	**1394,0**	**2275,0**	**11480,0**	**750447,0**
	m	14,165250	12,826602	12,700851	12,725588	12.505000
10	**t**	**1033,0**	**1128,0**	**1975,0**	**10781,0**	**691844,0**
	m	14,165250	12,826602	12,700851	12,725588	12.505000
15	**t**	**889,0**	**1071,0**	**2119,0**	**11055,0**	**634900,0**
	m	14,165250	12,826602	12,700851	12,725588	12.505000

Tab.2 - tabulka pro pvm - měřeno na clusteru hydra.fav.zcu.cz

N \ d / aw		0,05 / 0,1	0,01 / 0,1	0,01 / 0,05	0,005 / 0,01	0,001 / 0,01
2	**t**	**10,0**	**540,0**	**550,0**	**5830,0**	**532870,0**
	m	14,165250	12,826602	12,700851	12,725588	12,505000
5	**t**	**0,0**	**320,0**	**300,0**	**1830,0**	**235060,0**
	m	14,165250	12,826602	12,700851	12,725588	12,505000
10	**t**	**0,0**	**130,0**	**120,0**	**1020,0**	**109540,0**
	m	14,165250	12,826602	12,700851	12,725588	12,505000
15	**t**	**0,0**	**140,0**	**150,0**	**710,0**	**106150,0**
	m	14,165250	12,826602	12,700851	12,725588	12,505000

5.2 Vyhodnocení výsledků

Z tabulek je na první pohled zřejmé, že program v Javě (interpretovanému jazyku, navíc systému se sdílenou pamětí) je pomalejší, někdy až téměř řádově, než program napsaný v jazyce C a spuštěný na systému s distribuovanou pamětí.

5.2.1 Hmotnost

Co se týče přesnosti výsledků výpočtu hmotnosti, je vidět, že záleží pouze na **aw** a integračním kroku **d**. Čím jsou tyto hodnoty menší, tím je výsledek přesnější. Je to logické, neboť po čím větších částech se integruje, tím je výsledek nepřesnější. Navíc hraje i svou úlohu reálný typ double, který při svém zobrazování v počítači může způsobit malé nepřesnosti. Například při rozdělování kvádru mezi workery nebo počítání workerů s integračním krokem.

5.2.2 Čas výpočtu

Z obou tabulek je vidět, že **nejrychlejší** výpočty probíhaly při **vyšším** počtu workerů **N** a zároveň **největšími** měřenými kroky **d** i **aw**. Naopak **nejpomalejší** výpočty (kontrastuje s nejpřesnějšími výsledky) byly při **minimálním** počtu workerů a zároveň **nejkratším** kroku **d** a **aw**.

Se snižujícím se krokem **d** a **aw** se doba výpočtu zvyšuje velmi rapidně. Neboť při **50násobném** zmenšení kroku d se doba výpočtu prodloužila v průměru **600x** u **Javy** a u **pvm** dokonce řádově v násobku **10 000** až 100 000.

Zvýšení počtu workerů má za následek zrychlení výpočtu na obou systémech. U Javy není urychlení tak velké, při spuštění trojnásobku workerů se pohybuje maximálně kolem **1.5**, u pvm lze vyčíst z tabulky zrychlení maximálně kolem hodnoty **2.5** (rozdíl mezi 5 a 15 spuštěnými workery).

Integrační krok **d** (sám o sobě) nemá vliv na rychlost výpočtu na systému pvm, hodnoty ve druhém a třetím sloupci jsou podobné.

Stejně tak nemá na žádnou sledovanou veličinu vliv poměr (**d : aw**) při stejném počtu workerů; když se podíváme do tabulky, tak stejný poměr 1:2 mají sloupce 1 a 4, kde nelze vysledovat nějakou podobnost. Stejně tak poměr 1:10 u sloupců 2 a 5 nevykazuje podobné hodnoty.

5.2.3 Shrnutí

Pokud bychom chtěli najít kompromis mezi dosažením co nejpřesnějšího výsledku a co nejkratším časem výpočtu, zdá se být ideální zvolit **velký počet** workerů **N** a **nízké hodnoty** integračního kroku **d** i množství přidělené práce **aw**.

6 Závěr

Program je funkční v obou verzích, zadání úlohy bylo splněno. Oba programy vracejí stejné výsledky, co se týče vypočítané hmotnosti kvádru. Bohužel jsou některé hodnoty zatíženy chybou způsobenou použitím reálného typu double a jeho zobrazením v počítači. Nad vylepšením v tomto směru jsem již neuvažoval.

Možné rozšíření by bylo - zahrnout jiné funkce pro výpočet hustoty, případně i pro jiná tělesa.

Při spuštění pvm programu na clusteru Hydra jsem měl pouze ten problém, že ač jsem měl všechno nastaveno podle návodu uvedeném na stránkách předmětu (nahrát hostfile a program spustit v adresáři $HOME/pvm), program fungoval až po nakopírování do adresáře $HOME/pvm3/bin/LINUX. Tam ale běželo všechno již bez větších problémů.

K semestrální práci přikládám i vzorové konfigurační soubory v adresářích `config` a výstupy programů v podobě logovacích souborů (v adresářích `log`), které odpovídají použitým konfiguračním souborům.

7 Literatura

Herout, P.: Učebnice jazyka Java, Kopp 2003

Herout, P.: Učebnice jazyka C - 1. díl, Kopp 2004

http://hgf.vsb.cz/neu10/studium/pocitace/PVG/texty/1_2002/superpocitace/tema1/1f_paralelizace.htm

http://www.kiv.zcu.cz/studies/predmety/ppr/

Materiály k semestrální práci umístěné na stránkách předmětu

8 Příloha A - část zdrojového kódu - Java

Farmer.java - část:

```
/**
 * Metoda prideli cast prace danemu vlaknu workera k provedeni vypoctu dilci hmotnosti.
 * Metoda je synchronized, protoze prideleni prace workerovi musi byt atomicke.
 * */
synchronized private void assignToWorkerAt(int index) {

  // dokud je jeste co pocitat - nebylo prideleno celych 100% casti:
  if (this.assignedTotal < this.ASSIGNED_DONE) {

    // budu pocitat posledni kousek - dalsi uz presahuje kvadr
    if ((this.assignedTotal + this.zpracovani.getAssignedPart()) >= this.ASSIGNED_DONE) {
        // pridelim cast prace danemu vlaknu:
        this.workers[index].initWorkerData (this.assignedTotal *
                        this.zpracovani.getKvadr().getX(),
          0,0,  // 0 = fromY = fromZ -> sour. pocatku, pocita se po ctvercich podle x
          this.zpracovani.getKvadr().getX(),
          this.zpracovani.getKvadr().getY(),
          this.zpracovani.getKvadr().getZ(),        // koncove souradnice - posledni vypocet
          this.zpracovani.getDx(), this.zpracovani.getDy(),
          this.zpracovani.getDz(), this.zpracovani.getFunctionType());

        this.workers[index].setComputing();         // nastaveni stavu - spusti vypocet

        this.assignedTotal = this.ASSIGNED_DONE;    // prace skoncena - vsechno prideleno
    }
    else {                    // jeste nejsem na konci kvadru:
        // pridelim cast prace danemu vlaknu:
          this.workers[index].initWorkerData (this.assignedTotal *
                        this.zpracovani.getKvadr().getX(),
          0,0,  // 0 = fromY = fromZ -> sour. pocatku, pocita se po ctvercich podle x
                (this.assignedTotal + this.zpracovani.getAssignedPart()) *
                        this.zpracovani.getKvadr().getX(),
                this.zpracovani.getKvadr().getY(),
                this.zpracovani.getKvadr().getZ(),
                this.zpracovani.getDx(), this.zpracovani.getDy(),
                this.zpracovani.getDz(), this.zpracovani.getFunctionType());

        this.workers[index].setComputing();         // nastaveni stavu - spusti vypcet

        this.assignedTotal += this.zpracovani.getAssignedPart();// zvetsi se odpracovana cast
    }

  }
  else {
        // nechat skoncit vlakno daneho workera - neni co pocitat
        this.workers[index].endWork();
  }

}


/**
 * Metoda behu vlakna Farmera.
 * */
public void run() {
        super.run();

        boolean isWorkLeft = true;                          // zbyva prace na prideleni

        for (int i = 0; i < this.workers.length; i++) {           // spusteni vsech workeru
                //System.out.println("start");
                this.workers[i].start();
        }

        while (isWorkLeft) {                // dokud pocitam
                // pokud po cyklu for bude isWorkLeft false, vsechna vlakna jsou mrtva:
                isWorkLeft = false;

                for (int i = 0; i < this.workers.length; i++) {
```

```
                if (this.workers[i].getCurrentState() != Worker.DEAD) {  // vlakno
                            jeste nebylo umrtveno
                  //System.out.println(i + " jeste zije");
                  // zmenim na true - aspon jedno vlakno zije, cyklus while bude pokracovat:
                              isWorkLeft = true;
                }
                else {                                      // vlakno je mrtve
                     continue;                    // pokracuji testovani dalsiho vlakna
                }

        // testuji stav vlaken, pripadne jim pridelim praci:
        if (this.workers[i].getCurrentState() == Worker.IDLE) {    // prave bylo vytvorene
          this.assignToWorkerAt(i);                         // pridelim mu praci
        }
        if (this.workers[i].getCurrentState() == Worker.DONE) {    // prave dopocitalo
          //System.out.println(this.workers[i].getPartWeight());
          this.finalWeight += this.workers[i].getPartWeight();     // pricteni mezivysledku
          this.assignToWorkerAt(i);                                // pridelim dalsi praci
        }
      }
  }
  this.setNanosecs(System.nanoTime());      // zaznamenani casu konce vypoctu
  this.computing = false;                   // dopocitano - vsechna vlakna workeru mrtva

  System.out.println("farmer konci");
  //System.out.println("Vysledna hmotnost = " + this.finalWeight);
}
```

Worker.java - část:

```
/**
 * Metoda provede vlastni vypocet hmotnosti pozadovane casti kvadru podle
 * zadaneho vzorce (dx*dy*dz*ro(x+dx/2,y+dy/2,z+dz/2)). Pak nastavi stav
 * vlakna na DONE - ukoncen vypocet a vysledek lze prevzit.
 * */
public void computePartWeight() {

        // usek kvadru pres rozmer x:
        for (double iterX = this.fromX; iterX < this.toX; iterX += this.dx) {
                // usek kvadru pres rozmer y:
                for (double iterY = this.fromY; iterY < this.toY; iterY += this.dy) {
                        // usek kvadru pres rozmer z:
                        for (double iterZ = this.fromZ; iterZ < this.toZ; iterZ += this.dz) {
                                this.partWeight += this.dx * this.dy * this.dz * this.computeRo(
                                                                        iterX + this.dx/2,
                                                                        iterY + this.dy/2,
                                                                        iterZ + this.dz/2);
                        }
                }
        }
        this.currentState = DONE;    // vypocet skoncen, mezivysledek hotov, lze prevzit
}

/**
 * Metoda behu vlakna Workera.
 * */
public void run() {
        super.run();

        System.out.println("Worker " + this.myId + " bezi.");

        // dokud vlakno zije (dokud je pro nej nejaka prace):
        while (this.currentState != DEAD) {
                // Pokud ma vlakno nastavene hodnoty k pocitani (metodou initWorker()) ...
                if (this.currentState == WORKING) {
                        // Provede se vypocet:
                        this.computePartWeight();
                }
        }

        System.out.println("Worker " + this.myId + " konci.");
}
```

9 Příloha B - část zdrojového kódu - pvm

kvadr_farmer.c - část:

```
/* Funkce prideli cast kvadru danemu workerovi k provedeni vypoctu. */
void assign_to_worker_at(int index) {

  double from_x = assigned_total * kvadr.x;
  double from_y = 0.0;
  double from_z = 0.0;    // pocita se po "tenkych ctvercich" podle osy x
  double to_x;
  double to_y = kvadr.y;
  double to_z = kvadr.z;  /* nastaveni vsech hodnot pro workera */

  /* dokud je jeste co pocitat - nebylo prideleno celych 100% casti: */
  if (assigned_total < ASSIGNED_DONE) {
    /* budu pocitat posledni kousek - dalsi uz presahuje kvadr */
    if ((assigned_total + assigned_part) >= ASSIGNED_DONE) {

      to_x = kvadr.x;     /* donastaveni to_x - podle toho, zda je posledni */

      /* odeslani dat: */
      send_to_worker_at(index, from_x, from_y, from_z, to_x, to_y, to_z,
              function_type);

      assigned_total = ASSIGNED_DONE;   /* posledni kousek prace prirazen */
    }
    else {

      to_x = (assigned_total + assigned_part) * kvadr.x;  /* dalsi hodnota x */

      /* odeslani dat: */
      send_to_worker_at(index, from_x, from_y, from_z, to_x, to_y, to_z,
              function_type);

      assigned_total += assigned_part;  /* pricteni casti pridelene prace */
    }

    thread_count_running++;

  }
  else {

    pvm_send(index, DIE);      /* poslu zpravu, aby worker ukoncil cinnost */

  }

}


/* Funkce spusti pracovni proces farmera. (spousti se po init_farmer()) */
void run_farmer() {

  int i;                          /* promenna cyklu */
  int is_work_left = 1;           /* je jeste co pocitat; 0=false, 1=true */

  for (i = 0; i < thread_count; i++) {
    assign_to_worker_at(tids[i]); /* prideleni prace danemu workerovi */
  }

  while (is_work_left == 1) {     /* dokud je co pocitat */

    if (pvm_nrecv(-1, RESULT) > 0) {      /* dostal jsem vysledek*/

      /* Vyhodnoceni zaslaneho vysledku */
      pvm_upkint(&tid, 1, 1);
      pvm_upkdouble(&result_weight, 1, 1);

      sprintf(msg, "Prijata hmotnost od delnika %d: %f\n", tid, result_weight);
      printf(msg);
      log_to_file(msg);           /* vypis + logovani */

      final_weight += result_weight; /* pricteni vysledku k celkove hmotnosti */
      thread_count_running--;
```

```
      /* Pokud je jeste nejaka prace, pridelim workerovi dalsi cast: */
      assign_to_worker_at(tid);
    }

    /* Pokud neni co pocitat: */
    if ((thread_count_running <= 0) && (assigned_total >= ASSIGNED_DONE)) {
      is_work_left = 0;         /* neni prace - skonci cyklus */
    }

  } /* end of while */

  /* Ukonceni delniku */
  printf("Farmer: Ukoncuji delniky. Cekam na odezvu...\n");
  log_to_file("Farmer: Ukoncuji delniky. Cekam na odezvu...\n");  /* logovani */
  pvm_mcast(tids, thread_count, DIE);  /* broadcast zpravy s kodem KONEC */

  /* cekani na odezvu od delniku */
  for(i = 0; i < thread_count; i++) {

    pvm_recv(-1, DEAD);                 /* zprava od workera - mrtvy */
    pvm_upkint(&tid, 1, 1);             /* id delnika */

    sprintf(msg, "Farmer: Delnik cislo %d ukoncen\n", tid);
    printf(msg);
    log_to_file(msg);                   /* vypis + logovani */
  }

  pvm_exit();           /* konec farmera */

  stop_time = clock();   /* konec mereni casu vypoctu */
  final_time = (double)(stop_time - start_time) / (double)CLOCKS_PER_SEC * 1000;

  sprintf(msg, "Celkova hmotnost kvadru je: %f\n", final_weight);
  printf(msg);
  log_to_file(msg);                     /* vypis + logovani */
  sprintf(msg, "Celkova doba vypoctu je %6.5f ms.\n", final_time);
  printf(msg);
  log_to_file(msg);                     /* vypis + logovani */

  stop_log();                           /* ukonceni logovani do souboru */

  free ((void *) msg);
  free ((void *) tids);

}
```

kvadr_worker.c - část:

```
/* Funkce, ktera provede vypocet hmotnosti csati kvadru na zaklade
   predanych parametru a vrati vyslednou dilci hmotnost. */
double compute_part_weight(double from_x, double from_y, double from_z,
                      double to_x, double to_y, double to_z,
                      double dx, double dy, double dz,
                      int function_type) {

  double iter_x, iter_y, iter_z;      // promenne cyklu
  double part_weight = 0;             // vysledek vypoctu dilci hmotnosti

    /* usek kvadru pres rozmer x: */
              for (iter_x = from_x; iter_x < to_x; iter_x += dx) {
                      /* usek kvadru pres rozmer y: */
                      for (iter_y = from_y; iter_y < to_y; iter_y += dy) {
                              /* usek kvadru pres rozmer z: */
                              for (iter_z = from_z; iter_z < to_z; iter_z += dz) {

                                      /* pricteni spoctene casti hmotnosti podle vzorce: */
                                      part_weight += dx * dy * dz * compute_ro(
                                                            iter_x + dx/2,
                                                            iter_y + dy/2,
                                                            iter_z + dz/2, function_type);

                              }
                      }
```

```
              }

  current_state = DONE;               /* momentalni vypocet skoncil */

  return part_weight;
}

/* Hlavni funkce workera. */
int main(int argc, char **argv) {

  mytid = pvm_mytid();  /* prihlaseni do PVM */

  /* cekani na zpravu od farmera */
  pvm_recv(-1, IDLE);             /* zprava IDLE - worker vytvoren */
  pvm_upkint(&chief_tid, 1, 1);  /* zjisti cislo farmera*/

  current_state = IDLE;       /* worker vytvoren */

  /* kontrola zda je farmer ten pravy - totozny s otcem */
  if (chief_tid == pvm_parent()) {
    /* odpoved farmerovi */
    pvm_initsend(PvmDataDefault);/* inic. bufferu */
    pvm_pkint(&mytid, 1, 1);    /* ulozeni identifikace */
    pvm_send(chief_tid, RESPONSE);     /* vlastni odpoved farmerovi */
  }

  /* Dokud nebyl worker umrtven: */
  while (current_state != DEAD) {

              /* Pokud poslal farmer zpravu WORK: */
    if (pvm_nrecv(chief_tid, WORK) > 0) {

      part_weight = init_and_compute();   /* prevezme data a provede vypocet */

      /* Poslani vysledku */
      pvm_initsend(PvmDataDefault);  /* inic. bufferu */
      pvm_pkint(&mytid, 1, 1);
      pvm_pkdouble(&part_weight, 1, 1);
      pvm_send(chief_tid, RESULT);     /* poslani zpravy */

    }

    /* Pokud farmer posle ukoncovaci zpravu: */
    if (pvm_nrecv(chief_tid, DIE) > 0) {
      current_state = DEAD;
    }

       }  /* end of while */

  /* Zaslani zpravy farmerovi - worker uz je "mrtev". */
  pvm_pkint(&mytid, 1, 1);             /* ulozeni identifikace */
  pvm_send(chief_tid, DEAD);      /* poslu svuj stav - worker nezije */


  /* ukonceni cinnosti */
  pvm_exit();

  return 0;

}
```