Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky
Ústav Informatiky

DIPLOMOVÁ PRÁCA

Bratislava 2005

Šimon Zámečník

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky
Ústav Informatiky

DIPLOMOVÁ PRÁCA

Správa distribuovaných systémov

Dynamický load balancing pre PVM s využitím migrácie úloh

autor: Šimon Zámečník
dipl. vedúci: RNDr. Rastislav Královič, PhD.

Touto cestou by som sa chcel poďakovať môjmu diplomovému vedúcemu RNDr. Rastislavovi Královičovi, PhD. za jeho cenné rady a usmernenia pri písaní tejto práce, ako aj Fakulte sociálnych a ekonomických vied Univerzity Komenského v Bratislave za poskytnutie technického vybavenia pre potreby tejto práce.

Čestne prehlasujem, že som túto diplomovú prácu vypracoval sám a inú ako uvedenú literatúru som nepoužíval.

V Bratislave, 29. apríla 2005 ....

Obsah

Úvod

S nástupom rýchlejších a lacnejších počítačov sa v dnešnej dobe čoraz viac začína rozvíjať oblasť distribuovaného počítania a začína prenikať z výsostne akademického prostredia aj do oblasti komerčnej. S pomocou viacero menších počítačov spojených sieťou dnes možno postaviť paralelný stroj s výpočtovou silou podobnou veľkým superpočítačom a to za neporovnateľne nižšiu cenu.

Pod pojmom *paralelný distribuovaný systém* sa myslí systém skladajúci sa zo skupiny počítačov (nie nutne rovnakej architektúry) schopných navzájom komunikovať (napr. sieťou LAN alebo cez Internet), ktorý nám umožňuje spúšťať aplikácie súčasne na viacerých počítačoch. Vývoj v tejto oblasti viedol k množstvu výskumu rôznych problémov spojených s efektívnym využitím zdrojov, ktoré nám takéto paralelné systémy poskytujú. Jedným z týchto problémov je aj efektívna správa aplikácií bežiacich v takýchto systémoch. Množstvo prác sa zaoberalo *load balancingom*, teda vyrovnávaním záťaže a väčšina týchto prác je založená na snahe umiestniť úlohy pri ich spustení na jednotlivé uzly paralelného systému tak, aby boli jeho zdroje čo najefektívnejšie využité. Niektoré práce počítajú aj s možnosťou *migrácie úloh*, teda s presunom už spustenej úlohy z jedného počítača na druhý bez toho, aby boli výsledky výpočtov úlohy po jej presune stratené. Práve *migrácia úloh* nám pri vyrovnávaní záťaže dáva možnosť lepšie využiť zdroje hlavne v systémoch, kde sa zaťaženie počítačov dynamicky mení. Pojem *migrácie úloh* je známy už niekoľko rokov a bolo ukázané, že je naozaj zmysluplné ho pre load balancing použiť [Krueg88][Harch97].

Cieľom tejto práce je vytvoriť a experimentálne overiť heuristiku pre load balancing systému PVM, ktorá by využívala *migráciu úloh*. Systém PVM (Parallel Virtual Machine) poskytuje infraštruktúru pre pohodlné písanie a spúšťanie distribuovaných paralelných aplikácií. Sám o sebe však migráciu úloh nepodporuje, preto pre účely tejto práce používam jeho modifikáciu DPVM (Dynamite PVM) [Iskra00], ktorá migráciu úloh podporuje.

V prvej kapitole práce sú presnejšie vysvetlené základné pojmy ako load balancing a checkpointing, prehľad súčasných riešení vyrovnávania záťaže pre paralelné distribuované systémy ako aj rôznych prístupov k checkpointingu. V druhej kapitole je uvedený návrh novej heuristiky pre vyrovnávanie záťaže pre systém PVM, zdôvodnenie predpokladov pre jej prínos a popis jej praktickej implementácie pre systém DPVM. V tretej časti je táto heuristika empiricky porovnaná s pôvodnou heuristikou systému DPMV a uvedené výsledky tohto merania.

Kapitola 1

Prehľad problematiky

1.1 Základné pojmy

1.1.1 Distribuované počítanie

Distribuovaný systém je aplikácia skladajúca sa z viacerých komponentov bežiacich súčasne na rôznych počítačoch. Tieto počítače musia mať možnosť spolu komunikovať a byť schopné pracovať samostatne.

Paralelné počítanie je súčasné počítanie jednej úlohy na viacerých procesoroch za účelom zrýchlenia výpočtu. Procesorom môže byť buď CPU v počítači, alebo aj jeden uzol (počítač) v distribuovanom systéme.

Distribuované systémy sa využívajú pre paralelné počítanie v rôznych odvetviach vedy. V týchto systémoch sa úloha rozdelí na viacero podúloh, ktoré sú na sebe nezávislé, tieto sa potom počítajú súčasne na jednotlivých počítačoch a tým sa skracuje doba výpočtu. Takéto systémy sú buď *homogénne* alebo *heterogénne*:

- **homogénne** systémy sa skladajú z uzlov rovnakých parametrov - CPU, pamäť, OS a spravidla sú prepojené sieťou jedného typu
- **heterogénne** distribuované systémy môžu spájať počítače rôznych parametrov a tiež rôznych operačných systémov

Distribuovaný systém pre paralelné počítanie nazývame aj **počítačový cluster** a jednotlivé počítače v tomto clusteri nazývame **uzlami**.

1.1.2 PVM

PVM (Parallel Virtual Machine) [Geist94] je softwarový balík umožňujúci vytváranie heterogénnych počítačových clusterov spojením počítačov so systémom UNIX alebo Windows, navzájom prepojených sieťou. Tento systém je voľne dostupný, dobre prenositeľný a ľahko použiteľný, čo prispelo k jeho hojnému rozšíreniu pre vedecké využitie.

PVM je takzvaný *message passing* systém, teda systém, v ktorom sa paralelné úlohy synchronizujú a vymieňajú informácie posielaním správ. Naopak od *shared memory* systémov, kde majú paralelné úlohy prístup ku zdieľanej pamäti.

Samotný systém PVM sa skladá z démona, ktorý beží na každom uzle clustera a knižnice, ktorá sa prikompiluje k užívateľskému programu. Program využívajúci služby knižnice môže byť napísaný v rôznych programovacích jazykoch, najčastejšie sa využíva C, C++ a FORTRAN. Táto knižnica ponúka funkcie pre vytváranie nových úloh, ako aj pre komunikáciu medzi nimi. PVM démon potom zabezpečuje fyzické spúšťanie týchto úloh a prenos správ.

1.1.3 Load balancing

Pojem **load balancing** vo všeobecnosti označuje techniku vyrovnávania záťaže medzi viaceré zdroje, ako sú napríklad: procesory, počítače, disky atď.

V oblasti distribuovaného počítania označuje práve rovnomerné rozkladanie záťaže medzi jednotlivé uzly v clusteri. Load balancingom možno dosiahnuť efektívnejšieho využitia zdrojov v distribuovanom systéme a tým pádom aj skrátenie doby výpočtu jednotlivých úloh.

Keďže sa v počítačovom clusteri záťaž systému mení dynamicky (užívatelia spúšťajú a ukončujú úlohy), nevieme dopredu povedať, ako sa záťaž v systéme bude vyvíjať. Preto sú load balancingové algoritmy založené na heuristikách, ktoré sa viac alebo menej úspešne snažia predvídať, na ktorých uzloch záťaž stúpne, klesne a na ktorých sa prípadne nezmení.

Pre prehľad prístupov k load balancingu v distribuovaných systémoch pozri kapitolu 1.2.

1.1.4 Checkpointing

Checkpointing označuje techniku, ktorá umožňuje *zamrazenie* bežiacej úlohy tak, aby sa opäť dala uviesť do pôvodného bežiaceho stavu neskôr.

Táto technika má v praxi mnohé využitia, ako sú:

- zálohovanie prebiehajúcej úlohy pre prípad výpadku systému
- zálohovanie viacerých stavov úlohy pre prípad potreby navrátenia sa do niektorého z jej predošlých stavov
- presun úlohy z jedného počítača na druhý pre vyrovnávanie záťaže

Pre prehľad prístupov k checkpointingu a systémov, ktoré ho implementujú, pozri kapitolu 1.3.

1.2 Rôzne prístupy pre load balancing

V počítačových clusteroch často vzniká problém nevyrovnanej záťaže jednotlivých uzlov. Ten môže mať viacero príčin:

- Ak je cluster používaný aj na iné účely ako paralelné výpočty, môžu užívatelia spúšťaním svojich úloh saturovať záťaž na niektorých uzloch.
- V *message passing* systémoch je bežné, že niektorá úloha čaká, kým skončí iná a odovzdá jej výsledok. Ak sa na jednom uzle stretne viac úloh ktoré čakajú a na druhom zasa viac ktoré pracujú, bude druhý uzol preťažený.
- Nevyrovnaná záťaž môže byť spôsobená aj rozdielmi vo výkone jednotlivých uzlov heterogénneho clustera.

Tento problém sa snažíme riešiť load balancingom, teda vyrovnávaním záťaže. Väčšina výskumu v tejto oblasti sa zameriava na vyrovnávanie záťaže vo všeobecných distribuovaných systémoch a nie v message passing systémoch, ale ich prístupy sa dajú použiť aj pre clustre bežiace na message passing systéme.

Tieto prístupy sa dajú deliť podľa použitých informácií pri rozhodovaní na:

- statické
- dynamické

Ďalej podľa toho, ako sa snažia záťaž vyrovnať:

- len efektívnym prideľovaním nových úloh na uzly
- aj migráciou už bežiacich úloh

1.2.1 Statický prístup

Statický prístup k load balancingu sa snaží efektívne rozdeľovať nové úlohy medzi jednotlivé uzly na základe statických informácií o štruktúre distribuovaného systému. Tieto informácie väčšinou zahŕňajú topológiu systému, relatívny výkon jednotlivých uzlov, a niekedy aj štatistické informácie o záťaži systémov (tieto však nie sú získavané online, ale napr. ručným meraním).

V praxi je tento prístup použiteľný pre systémy, kde sa záťaž uzlov len málo mení, alebo sa dá dobre predpovedať, v systémoch, ktoré používa len jeden užívateľ, prípadne len jedna aplikácia. Nepoužiteľným sa stáva pri systémoch, kde sa záťaž dynamicky mení.

Jeho výhodou je, že pri jeho aplikácii nevzniká žiaden overhead pre zber dynamických informácií o stave systému, nakoľko ich netreba.

1.2.2 Dynamický prístup

Prístup, ktorý pre vyrovnávanie záťaže využíva aktuálne dynamické dáta získané monitorovaním systému, sa nazýva **dynamický**. Tieto dáta sú potom využité pri prideľovaní jednotlivých úloh uzlom.

Takýto systém sa vie oveľa pružnejšie prispôsobiť zmenám v záťaži a intuitívne je jasné, že je efektívnejší než statický prístup. Jeho nevýhodou je, opačne než u statického prístupu, overhead vzniknutý monitorovaním systému.

1.2.3 Load balancing prideľovaním nových úloh

Tento prístup používa väčšina load balancingových systémov. Snaží sa novopríchodzie úlohy (také, ktoré sa majú spustiť) efektívne rozmiestniť na jednotlivé uzly. Pri tomto výbere môže, a nemusí použiť predošlé informácie o úlohe.

Nevýhodou tohto prístupu je to, že ak pridelíme nejakú úlohu na uzol, ktorý je preťažený, bude táto úloha spomaľovaná až do svojho ukončenia, nakoľko ju systém nemá ako presunúť inam. Toto spomalenie bude tým výraznejšie, čím dlhší život táto úloha má.

Kvôli tomu sa väčšina týchto heuristík snaží odhadnúť alebo inak predpovedať dĺžku života úlohy ešte pred jej spustením. To sa robí v mnohých prípadoch na základe pozorovania predošlých úloh rovnakého typu. Zaujímavý prístup implementovali Jiubin Ju, Gaochao Xu a Kun Yang [Ju95]. Pri zisťovaní správania sa úlohy ju najprv na krátky čas spustia, sledujú funkcie ktoré používa a na základe toho sa rozhodnú, či úlohu nechajú bežať, alebo ju ukončia a spustia znova na inom uzle.

1.2.4 Load balancing využívajúci migráciu úloh

Ak pridáme do systému možnosť migrácie úloh, dostávame tým možnosť riešiť problém, ktorý vzniká pri pridelení úlohy na preťažený uzol. V takom prípade ju jednoducho presunieme na uzol iný.

Pri migrácii úlohy však vždy vzniká overhead, s ktorým treba rátať. Ak by sme napríklad zmigrovali úlohu tesne pred jej skončením, nič by sme tým nezískali, pretože za čas, za ktorý je úloha migrovaná, by už bola skončila a my sme ju tým vlastne zdržali.

Pri vyrovnávaní záťaže pomocou migrácie úloh sa väčšinou uplatňujú tri základné kroky:

- výber zdroja migrácie
- výber úlohy, ktorú chceme migrovať
- a výber cieľa, kam chceme úlohu presunúť

Tieto kroky nemusia nutne nasledovať v tom istom poradí, a preto ďalej rozdeľujeme prístupy na:

- **sender-initiated** - pri týchto prístupoch sa preťažené uzly snažia zbaviť záťaže a hľadajú voľné uzly, čo je dobré hlavne v systémoch, kde je len málo preťažených uzlov.
- **receiver-initiated** - tu zasa naopak uzly, ktoré nemajú čo robiť, hľadajú preťažené. Táto technika je efektívnejšia v systémoch, v ktorých je zasa len málo voľných uzlov.

- **symetrické** - spájajú oba predchodzie prístupy a snažia sa spojiť ich výhody. Sú použiteľnejšie na širšie spektrum systémov než obe predchádzajúce.
- **náhodné** - u týchto sa cieľ pre migráciu vyberá náhodne. To, že táto jednoduchá technika má opodstatnenie, ukázal okrem iných aj Kunz [Kunz91].

S príchodom nového a rýchlejšieho hardvérového vybavenia sa cena za migrovanie úloh stále znižuje a preto je aj efektívnejšie ju pre load balancing použiť.

1.3 Prehľad checkpointovacích systémov

Myšlienka checkpointingu je známa už mnoho rokov a viaceré práce ju dokázali úspešne implementovať. Aj napriek tomu sa zatiaľ v praxi táto technika neuchytila, no podľa [Miloj00] je len otázkou času, kedy sa jej používanie rozšíri.

Checkpointing, ako už bolo spomenuté, je technika, ktorá umožňuje *zamraziť* úlohu a jej stav a potom ju neskoršie opäť spustiť tak, aby pokračovala v tom istom stave ako pred zmrazením.

Proces migrácie pozostáva z troch krokov:

- **zaznamenanie stavu úlohy** - stav úlohy treba zaznamenať do formy vhodnej na prenos (napr. do súboru). Stav by mal zahŕňať pamäť, ktorú úloha používa, informácie o zdrojoch, ktoré používa (súbory, sieťové spojenia) a údaje o jej súčasnom stave (napr. obsah CPU registrov)
- **prenos checkpointu** - po zamrazení treba úlohu fyzicky preniesť na cieľový uzol, ak je iný ako zdrojový
- **znovuspustenie** - zahrňuje znovuspustenie úlohy na cieľovom uzle a obnovenie jej pôvodného stavu

Pri tomto procese sa stretávame s mnohými problémami, ako sú:

- **otvorené súbory** - úloha môže čítať alebo zapisovať do súborov, ktoré po jej znovuspustení nemusia byť dostupné

- **zdieľané knižnice** - úloha môže využívať aj zdieľané knižnice, ktoré taktiež nemusia byť po jej znovuspustení dostupné, prípadne sú inej verzie ako pôvodné
- **komunikácia s inými procesmi** - ak úloha komunikovala s inými, nemusia byť tieto po jej znovuspustení aktívne
- **heterogénnosť prostredia** - ak úlohu prenášame z jednej počítačovej architektúry na druhú, tak môžu mať tieto architektúry napríklad rozdielny spôsob ukladania dát v pamäti

Checkpointing môžeme z hľadiska spolupráce migrovanej úlohy deliť na:

- **s podporou migrovanej úlohy** - ak úloha spolupracuje pri procese migrácie, či už poskytnutím údajov o svojom stave, alebo informovaním o čase, kedy je ju možné zmigrovať
- **transparentný** - systém zmigruje úlohu bez toho, aby o tom úloha vedela, alebo tomu akokoľvek napomohla

Podľa toho, kde je samotný mechanizmus migrácie implementovaný, poznáme checkpointing:

- **kernel level** - tento je implementovaný priamo v jadre operačného systému. Dáva mu to väčšiu flexibilitu pri zisťovaní stavu úlohy, ako aj vyšší výkon pri migrácii, ale na druhej strane býva do veľkej miery závislý na druhu a verzii operačného systému
- **user level** - tento pracuje len na užívateľskej úrovni. Jeho nevýhodou je ťažšie získavanie informácií o úlohe, ale nie je až tak závislý na operačnom systéme

V nasledujúcich podkapitolách sú uvedené postupy týchto checkpointerov pri riešení problémov migrácie a systémy, ktoré ich implementujú. Pre rozsiahlejší prehľad o dostupných technikách a systémoch pre migráciu úloh pozri [Miloj00].

1.3.1 Checkpointing s podporou migrovanej úlohy

Tento prístup je z hľadiska programovania najjednoduchší, nakoľko nám úloha napomáha pri procese migrácie. Sama napríklad rozhodne, v ktorých miestach jej programu ju možno zmigrovať, alebo nám poradí, ktoré oblasti pamäte používa, a ktoré nie.

Ak je v úlohe miesto, kde je vhodné migráciu vykonať (tam, kde úloha nemá otvorené súbory, prípadne používa najmenej pamäte), úloha to sama ohlási zavolaním špeciálnej funkcie. Checkpointer potom vie, že môže úlohu v tomto mieste efektívne zmigrovať.

Ďalším vylepšením, ktoré môže aplikácia poskytnúť, je informácia o používanej pamäti. Stačí, aby úloha vyznačila sekcie, v ktorej danú časť pamäte používa a checkpointeru potom pri migrácii stačí preniesť len oblasti, ktoré sú označené ako používané.

Obe tieto techniky sú používané v systéme **libckpt** [Plank95]. Ďalším príkladom pre netransparentný checkpointing je **CUMULVS** [Kohl98].

Aj keď je tento prístup relatívne jednoduchý na implementáciu, má tú nevýhodu, že potrebujeme zmeniť zdrojový kód úlohy, pre ktorú chceme pridať možnosť migrácie. To však nemusí byť vo všeobecnosti možné, nakoľko nemusíme mať prístup ku zdrojovému kódu danej aplikácie.

1.3.2 Transparentný Checkpointing

Pri tomto spôsobe checkpointingu sa deje migrovanie aplikácie bez jej vedomia. Výhodou síce je, že migrované aplikácie nemusíme meniť, ale na druhej strane je programátorsky omnoho náročnejší na implementáciu.

Existujú rôzne systémy, ktoré dosahujú transparentnosť len čiastočne, napríklad aj systém **DPVM** [Iskra00]. Na to, aby sa dali v DPVM úlohy migrovať, musia byť pri kompilácii prilinkované k upravenému dynamickému loaderu, ktorý potom zaobstaráva réžiu okolo checkpointingu. Samotný zdrojový kód úlohy ale meniť netreba.

1.3.3 Kernel level checkpointing

Problémy, ktoré treba pri migrácii úloh riešiť, ako napríklad zisťovanie zoznamu otvorených súborov alebo aktívnych sieťových spojení, sa dajú relatívne ľahko riešiť priamo v jadre operačného systému. Takýto checkpointer sa jednoducho zavesí na systémové služby poskytované jadrom (napr. otváranie a zatváranie súborov) a takto získa informácie o stave úlohy. Nakoľko pracuje v *kernel móde*, má prístup k interným štruktúram jadra a môže tak získať informácie o migrovanej úlohe omnoho jednoduchšie než program, ktorý beží v *user móde*.

Takéto checkpointery sú väčšinou súčasťou celého distribuovaného operačného systému, ako je napríklad **MOSIX** [Barak98], alebo **Sprite** [Oust88].

1.3.4 User level checkpointing

Ak používame komerčný operačný systém alebo z nejakého dôvodu nechceme zasahovať do jeho jadra, môžeme použiť checkpointing, ktorý je implementovaný v užívateľskom móde. Tieto systémy sú ľahšie prenositeľné, nakoľko využívajú pre zisťovanie informácií o stave úlohy len štandardné rozhrania, ktoré sú dostupné z užívateľského módu daného operačného systému.

Toto však zistiteľné údaje o úlohe obmedzuje, prípadne ich získavanie sťažuje. Preto sú tieto systémy vo všeobecnosti menej výkonné než tie, ktoré bežia na úrovni jadra.

Príkladom takýchto *user level* checkpointerov sú už spomínané **DPVM** a **libckpt**.

1.3.5 Ďalšie použité techniky pre zlepšenie výkonu

Existujú viaceré techniky ako skrátiť čas potrebný pre migráciu úlohy. Väčšinou sa hlavne snažíme zmenšiť objem dát, ktoré treba preniesť na cieľový uzol, prípadne urýchliť proces prenosu.

Ak máme prístup k presným informáciám o stave pamäte úlohy, môžeme do obsahu checkpointovacieho súboru zahrnúť len pamäťové stránky, ktoré boli od spustenia úlohy modifikované. Táto technika sa dá využiť aj pre takzvaný *inkrementálny* checkpointing, kedy zapisujeme len stránky, ktoré sa zmenili od posledného checkpointu. Taktiež je použitá v systéme používajúcom *precopy*, kde sa celý obsah pamäte úlohy presunie na cieľový uzol, ale počas tohto prenosu sa ponechá úloha v behu. Po dokončení prenosu sa úloha ukončí a prenesú sa už len stránky, ktoré boli od počiatku prenosu zmenené. Toto síce prináša ďalší overhead pri migrácii úlohy, ale minimalizuje čas prerušenia behu úlohy. Tento prístup používa **V Kernel** [Cher88].

Ďalšou z možností je online komprimovanie obsahu checkpointovacieho súboru. Plank et al. [Plank95a] použil stratégiu, kedy sa komprimujú len časti zmenených stránok, ktoré boli úlohou zmenené od posledného checkpointu.

Kapitola 2

Heuristika pre load balancing

Systém PVM je napriek svojmu veku stále obľúbený, hlavne v akademickej sfére. Je to spôsobené jednoduchosťou programového rozhrania a podporou viacerých sofvérových aj hardvérových platforiem. V dnešnej dobe existuje množstvo nevyužitej výpočtovej sily v mnohých organizáciách a práve PVM nám dáva možnosť ju použiť v náš prospech. Pre skutočne efektívne využitie je ale potrebné implementovať aj systém, ktorý by dokázal vyrovnávať záťaž aj tam, kde sa dynamicky mení pod vplyvom ostatných užívateľov.

Táto kapitola predstavuje heuristiku využívajúcu niektoré špecifické vlastnosti tohto systému pre riešenie úlohy vyrovnávania záťaže centrálne riadenou migráciou úloh. Jej použitie sa však nevzťahuje len na PVM, ale v podstate na všetky podobné *message passing* paralelné systémy schopné vykonávať migráciu.

2.1 Motivácie a predpoklady pre navrhovanú heuristiku

Dynamickému load balancingu s využitím migrácie úloh v počítačových clusteroch bolo venované množstvo výskumu. Väčšina týchto prác sa však zameriavala na vyrovnávanie záťaže v bežných viacužívateľských systémoch, kde sa spúšťa množstvo samostatných úloh (ako napríklad kompilácie zdrojových súborov).

Tieto práce zistili, že pre vhodnosť úlohy pre migráciu je hlavný jej vek [Harch97]. Z pozorovania reálnych systémov bolo zistené, že predpokladaný zostávajúci čas behu úlohy sa zvyšuje lineárne s jej vekom. V takomto systéme, v ktorom je väčšina úloh krátkych, je naozaj žiaduce migrovať staršie úlohy, pretože overhead vzniknutý pre takúto úlohu pri jej migrácii je v porovnaní s jej dlhým vekom zanedbateľný.

Ak sa však pozrieme na systém, na ktorom beží PVM, je situácia trochu iná. V tomto systéme je typické pre úlohu, ktorá bola spustená jej rodičom, že jej dĺžka života je skoro taká istá ako dĺžka života jej rodiča. PVM aplikácie totiž spustia niekoľko potomkov a potom im posielajú prácu pomocou správ. Ak rodič zistí, že jeho potomok dopočítal úlohu, ktorá mu bola pridelená, neukončí ho, ale snaží sa mu väčšinou priradiť prácu ďalšiu, ak samozrejme ešte nejaká zvýšila. Ak by ho totiž po výpočte ukončil a na druhý výpočet by spustil ďalšiu inštanciu, bolo by to plytvanie, nakoľko spustenie novej inštancie (navyše na nelokálnom uzle) so sebou prináša overhead.

Toto ale prakticky znamená, že v systéme PVM žije väčšina úloh relatívne dlho. Preto v tomto kontexte stráca vek úlohy ako kritérium pre migráciu zmysel a je potrebné vykonávať výber úloh iným spôsobom.

Nakoľko je systém PVM založený na posielaní správ, jednotlivé úlohy buď počítajú, alebo posielajú či prijímajú správy. Ak úloha čaká na príjem správy, nezaťažuje iné úlohy bežiace na tom istom uzle. Pri výbere úloh pre migráciu v systéme PVM by preto bolo dobré, ak by sme vedeli, ako sa bude daná úloha správať v budúcnosti.

Ďalšou dôležitou črtou systému PVM je to, že sa väčšinou spúšťa viacero inštancií tej istej úlohy. Ak by sme napríklad chceli implementovať aplikáciu pre počítanie fraktálov, vytvorili by sme jednu hlavnú úlohu, ktorá by spustila viac inštancií jednej pracovnej úlohy a každej z tejto inštancií by dala vypočítať časť z výsledného obrázku. Je tu teda predpoklad (aj keď to nemusí byť pravidlom), že tie isté úlohy sa budú správať podobne. Dĺžka života aplikácie sa síce môže meniť v závislosti na dátach, nad ktorými pracuje, ale v mnohých prípadoch je jej spávanie podobné, teda pomer medzi časom stráveným výpočtom a časom stráveným čakaním je rovnaký.

Z predchádzajúceho vyplýva, akým spôsobom by bolo možné predpovedať správanie sa úlohy v systéme PVM. Ak by sme mali dostupné informácie z predošlých spustení aplikácie, mohli by sme na ich základe predpovedať vývoj práve prebiehajúcej. Zber údajov by sa dal zabezpečiť pozorovaním spustených úloh a zápisom údajov o ich behu do nejakej centrálnej štatistiky po ich skončení.

V predchádzajúcich prácach bola snaha využiť štatistické informácie o predošlom behu úloh pri všeobecných viacužívateľských systémoch, no počet rôznych úloh je v tomto systéme veľmi veľký a navyše sa na rôznych vstupných dátach môžu správať rozlične. Systém PVM však dáva predpoklad pre fungovanie takéhoto prístupu, nakoľko počet PVM aplikácií použitých na jednom systéme nie je až taký veľký.

2.2 Navrhovaná heuristika

Táto heuristika predpokladá distribuovaný systém, v ktorom bežia dva typy úloh, tie, ktoré sú migrovateľné a tie, ktoré nie. Ďalej počíta s centrálnym riadením, teda že existuje uzol, na ktorom sa zbiehajú všetky informácie o stave systému a odkiaľ sa riadia všetky migrácie. Daný algoritmus potom beží priamo na centrálnom uzle.

Skôr než uvediem samotnú heuristiku, je potrebné vysvetliť vstupy, s ktorými pracuje a ako ich môžeme vypočítať.

2.2.1 Použité vstupy

Pre výpočet aktuálneho percentuálneho využitia procesora úlohou, teda pomer času, ktorý úloha využila a mala možnosť využiť, je použitý vzťah (2.1)

$$LOAD\_ACT = \frac{(utime + stime)/HZ}{elapsed\_time} * host\_load\_avg \tag{2.1}$$

kde $utime$ a $stime$ sú počty hodinových taktov, ktoré úloha strávila behom vo svojom programe a v systémových obsluhách od posledného merania, HZ je frekvencia hodín na danom procesore, $elapsed\_time$ je čas od posledného merania v sekundách a $host\_load\_avg$ je priemerný počet aktívnych úloh na danom procesore za poslednú minútu.

Ďalej budeme potrebovať aj celkové priemerné využitie procesora úlohou, teda za celý jej život, to vypočítame vždy pri novom meraní podľa vzťahu (2.2)

$$AVG_i = \frac{AVG_{i-1} * total\_elapsed_{i-1} + LOAD\_ACT * elapsed\_time}{total\_elapsed_{i-1} + elapsed\_time} \tag{2.2}$$

kde AVG_i je priemer využitia procesora po *i-tom* meraní, $total\_elapsed_i$ je celkový čas v sekundách od začiatku behu úlohy až po *i-te* meranie, $LOAD\_ACT$ je využitie vypočítané zo vzťahu (2.1) a $elapsed\_time$ je čas ubehnutý od posledného merania taktiež v sekundách.

Pre update priemerného využitia procesora úlohou v histórii behu úloh použijeme po jej ukončení nasledovný vzťah (2.3)

$$S\_AVG_i = \frac{S\_AVG_{i-1} * (i-1) + AVG}{i} \tag{2.3}$$

kde S_AVG_i je priemer využitia procesora po *i-tom* zázname do histórie meraní danej úlohy a AVG je priemerné využitie procesora za celý beh úlohy.

Podobne ako pre S_AVG vypočítame aj S_LIFE, teda priemernú dĺžku života úlohy v štatistike histórie behu úloh.

Na výpočet predpokladaného využitia procesora úlohou pomocou histórie behu úloh použijeme vzťah (2.4)

$$LOAD\_EXP = \frac{S\_AVG/S\_LIFE - AVG/total\_elapsed}{S\_LIFE - total\_elapsed} \tag{2.4}$$

kde S_AVG je najaktuálnejší záznam priemerného využitia procesora v histórii behu úloh pre danú úlohu, S_LIFE je najaktuálnejší záznam priemernej dĺžky života v histórii behu úloh pre danú úlohu a $total\_elapsed$ je celková dĺžka života úlohy od jej spustenia.

Taktiež môžeme vypočítať predpokladanú záťaž daného uzla pomocou vzťahu (2.5)

$$LOAD\_EXP\_HOST = LOAD\_NONPVM + \sum LOAD\_EXP_T \tag{2.5}$$

kde $LOAD\_NONPVM$ je záťaž uzla, ktorá je spôsobená úlohami, ktoré nepatria PVM, alebo sa nedajú migrovať a $\sum LOAD\_EXP_T$ je suma predpokladaných záťaží všetkých takých úloh T, ktoré patria danému uzlu a sú migrovateľné.

Teraz môžeme popísať základné vstupy pre algoritmy 1 a 2, ktorými sú polia **Host[]** a **Task[]**. Tieto polia obsahujú údaje o všetkých uzloch a úlohách v systéme.

Každá položka v poli **Task[]** obsahuje nasledovné údaje o úlohách:

- **LOAD_ACT** - toto je aktuálne percentuálne využitie procesora úlohou, vypočíta sa na základe vzťahu (2.1)

- **LOAD_AVG** - priemerné percentuálne využitie procesora úlohou za celý čas jej behu podľa vzťahu (2.2)
- **LOAD_EXP** - označuje predpokladané využitie procesora úlohou podľa vzťahu (2.4)
- **TOTAL_LIFE** - celkový čas behu úlohy
- **MIGRATABLE** - označuje, či je úloha migrovateľná

Pole **Host[]** obsahuje nasledovné údaje o každom uzle:

- **LOAD_ACT** - celkové aktuálne zaťaženie daného uzla, je to vlastne priemerný počet aktívnych úloh za poslednú minútu
- **LOAD_PVM** - aktuálna záťaž, ktorú spôsobujú migrovateľné úlohy
- **LOAD_NONPVM** - aktuálna záťaž, ktorú spôsobujú úlohy, ktoré buď nepatria PVM, alebo sú nemigrovateľné
- **LOAD_EXP** - označuje predpokladanú záťaž uzla vypočítanú podľa vzťahu (2.5)
- **TREND** - rozdiel $LOAD\_ACT$ od posledného merania

2.2.2 Vysvetlenie algoritmu

Algoritmus 2 je hlavným telom celého load balancingu. Na jeho vstupe máme všetky informácie o uzloch a úlohách v systéme podľa podkapitoly 2.2.1.

Algoritmus v prvom kroku vyhľadáva uzly, ktorých zaťaženie prekročilo $LOAD\_TRESHOLD$. Táto hodnota je daná pevne a určuje hranicu, od ktorej sa uzol považuje za preťažený a tým pádom aj ako vhodný kandidát pre zdroj migrácie. Ak by táto hodnota bola príliš nízka, heuristika by považovala príliš veľa uzlov za preťažené a vykonávala by zbytočné migrácie. Ak by naopak bola hodnota $LOAD\_TRESHOLD$ príliš vysoká, systém by vykonával migrácií málo na to, aby to pozitívne dokázalo ovplyvniť vyrovnanie záťaže v systéme. V mojej heuristike je $LOAD\_TRESHOLD = 2$. Táto hodnota bola zistená empiricky v [Marei02]. Taktiež vyberá len uzly, ktoré ešte v tomto behu heuristiky neboli použité ako cieľ pre migráciu, nakoľko chceme zabrániť

Vstup: zdrojový uzol **S**, cieľový uzol **D**, úloha **T**
Vstup: údaje o bežiacich procesoch **Task[]**
Vstup: údaje o uzloch v systéme **Host[]**

Výstup: numerické skóre pre danú migráciu, čím vyššie, tým lepšie

if *Hosts[**D**].LOAD_EXP + Task[**D**].LOAD_EXP + MIN_IMPROVE + Host[**D**].TREND > Host[**S**] - Task[**T**].LOAD_ACT* **then**
    return *-1*;
end
S_load_new := Host[**S**].LOAD_EXP - Task[**T**].LOAD_EXP + Host[**S**].TREND;
if *S_load_new < 0* **then**
    S_load_new := 0
end
D_load_new := Host[**D**].LOAD_EXP + Task[**T**].LOAD_EXP + Host[**D**].TREND;
if *D_load_new < 0* **then**
    D_load_new := 0
end
score := $1 - (\frac{S\_load\_new + D\_load\_new}{LOAD\_TRESHOLD * 2})/MAX\_CUR\_LOAD$
return *score*;

Algoritmus 1: výpočet skóre pre migráciu úlohy **T** z uzla **S** na uzol **D**

stavu, kedy by sme nejakú úlohu niekam zmigrovali a hneď by sme ju vzápätí presunuli inam.

Po výbere kandidátov pre zdroj migrácie potom pre každý takýto uzol, všetky migrovateľné úlohy na ňom a všetky potenciálne cieľové uzly určí ich skóre pomocou algoritmu 1. V tomto kroku taktiež vyberá cieľové uzly len tie, ktoré nemajú záťaž väčšiu než $LOAD\_TRESHOLD$ a neboli ešte v tomto behu heuristiky zdrojom migrácie.

Spomedzi všetkých ohodnotení kombinácií teraz vyberie takú, ktorej skóre je maximálne a kladné. Ak sme takú našli, vykonáme migráciu úlohy **T** z uzla **S** na uzol **D**, upravíme polia **Host[]** a **Task[]** podľa vykonanej migrácie a pokračujeme v hľadaní ďalšej možnej migrácie. V jednom behu ale nevykonáme viac než $MAX\_MIGRATIONS$ migrácií, nakoľko s každou ďalšou migráciou sa stávajú údaje v poliach **Host[]** a **Task[]** menej spoľahlivé. V mojej heuristike je položený parameter

Vstup: údaje o uzloch v systéme **Host[]**
Vstup: údaje o úlohách v systéme **Task[]**

migrations_left := MAX_MIGRATIONS;
while *migrations_left > 0* **do**
    max_score := 0;
    for *každý uzol **S** pre ktorý Host[**S**].LOAD_ACT > LOAD_TRESHOLD a ešte nebol cieľom migrácie* **do**
        for *každá migrovateľná úloha **T** na uzle **S*** **do**
            for *každý uzol **D** iný od **S** ktorý nebol ešte zdrojom migrácie a pre ktorý Host[**D**].LOAD_ACT < LOAD_TRESHOLD* **do**
                vypočítaj skóre pre zdroj **S**, cieľ **D** a úlohu **T** pomocou algoritmu 1
                if *vypočítané skóre > max_score* **then**
                    max_score := vypočítané skóre;
                    BESTT := **T**;
                    BESTS := **S**;
                    BESTD := **D**;
                end
            end
        end
    end
    if *max_score > 0* **then**
        vykonaj migráciu úlohy **BESTT** z uzla **BESTS** na uzol **BESTD**;
        migrations_left - -;
        uprav polia Host[] a Task[] podľa vykonanej migrácie;
    else
        migrations_left := 0;
    end
end
zaspi na SCHED_WAIT sekúnd;
opakuj od začiatku;

Algoritmus 2: Heuristika pre dynamický load balancing

$MAX\_MIGRATIONS = 4$ a je do istej miery závislý na systéme, kde heuristiku používam. Ak by sme mali systém, kde je množstvo počítačov, mohla by byť táto hodnota vyššia.

Ak sa spomedzi ohodnotených kombinácií nenašla žiadna taká, ktorá by mala kladné skóre, alebo už bol vykonaný maximálny počet migrácií, tak algoritmus čaká $SCHED\_WAIT$ sekúnd, kým sa nespustí znova. V mojom algoritme je $SCHED\_WAIT = 60$ preto, aby sa medzi jednotlivými behmi balancera stačili aktualizovať informácie obsiahnuté v poliach **Host[]** a **Task[]**.

Algoritmus 1 počíta numerické skóre pre danú kombináciu zdrojového uzla **S**, cieľového uzla **D** a úlohy **T**. Snaží sa číselne vyjadriť preferenciu prípadnej migrácie úlohy **T** z uzla **S** na uzol **D**. Tento algoritmus už pracuje s predpokladanými záťažami, ktoré sú vypočítané na základe histórie behu úloh.

V prvom kroku overí, či by presun úlohy z **S** na **D** nespôsobil to, že situácia na **D** bude po presune ešte horšia než nová situácia na **S**. Používa k tomu ešte aj konštantu $MIN\_IMPROVE$, ktorá udáva, o koľko by sa mala situácia na **D** oproti **S** zlepšiť. Teda po migrácii by mala záťaž na **D** byť aspoň o $MIN\_IMPROVE$ menšia než záťaž na **S**. Túto konštantu som položil $MIN\_IMPROVE = 0.5$ a vyjadruje svojím spôsobom cenu migrácie. Ak teda zistí, že by nastalo zhoršenie situácie, tak ako skóre vráti -1.

Po prvom kroku ďalej vypočíta predpokladané záťaže na uzloch **S** a **D** po migrácii úlohy **T**. Od predpokladanej záťaže **S** odpočíta predpokladanú záťaž úlohy **T** a ešte pridá hodnotu $TREND$ pre **S**. Pomocou $TREND$ sa snaží vykompenzovať nepresnosť údajov o zaťažení uzlov, nakoľko sú tieto získavané ako priemerné zaťaženia za poslednú minútu. $TREND$ je vlastne rozdiel aktuálnej priemernej záťaže uzla a priemernej záťaže uzla z posledného merania. Ak je väčší než 0, znamená to, že záťaž bude na uzle pravdepodobne stúpať, ak je naopak menší než 0, má záťaž tendenciu klesať. S predpokladanou záťažou na **D** urobí to isté čo pri **S**, ale namiesto toho, aby záťaž úlohy **T** odpočítal, ju pripočíta, nakoľko má byť táto úloha na uzol **D** presunutá.

Obe tieto predpokladané záťaže potom vyjadrí v percentách hodnoty $LOAD\_TRESHOLD$ a celkové skóre potom vypočíta ako 1 - aritmetický priemer týchto percentuálnych hodnôt normovaný podľa hodnoty MAX_CUR_LOAD, ktorá je rovná maximálnej aktuálnej záťaži spomedzi všetkých záťaží uzlov. Toto normovanie je na to, aby aj pri vysokých záťažiach nevychádzali záporné ohodnotenia. Odpočítavanie od 1 sa robí preto, aby platilo, že čím vyššie skóre, tým lepšie. Nakoľko tu od 1 nikdy neodpočítame zápornú hodnotu[1], bude maximálne možné skóre 1.

2.3 Praktická implementácia

Pre praktické zhodnotenie navrhovanej heuristiky bolo potrebné ju implementovať na nejaký reálny PVM systém, ktorý by podporoval migráciu úloh. Po zhodnotení viacerých systémov bol ako vhodný vybraný DPVM [Iskra00].

2.3.1 Systém Dynamite PVM

Systém Dynamite PVM sa skladá z *Dynamite checkpointera*, ktorý zabezpečuje všeobecnú migráciu úloh, ďalej *DPVM*, čo je upravená verzia PVM, ktorá podporuje migráciu úloh a napokon *monitora*, ktorý sa stará o load balancing.

Dynamite checkpointer je systém umožňujúci migráciu úloh v operačných systémoch *Linux* a *Solaris*. Je to user level checkpointer a je implementovaný ako alternatívny dynamický zavádzač glibc[2] pre ELF binárne súbory. Dynamický zavádzač (dynamic loader) je knižnica, ktorá sa pri linkovaní pridá k užívateľskému programu a stará sa o korektné spustenie úlohy v systéme a nahrávanie dynamických knižníc potrebných pre beh aplikácie. Ak je Dynamite checkpointer prilinkovaný k programu, tak sa pri jeho spustení zavesí na rôzne systémové obsluhy (otváranie a zatváranie súborov atď.), aby tak mohol sledovať, ktoré zdroje úloha používa. Po poslaní signálu *SIGUSR1* takejto úlohe ju checkpointer preruší a zapíše na disk checkpoint súbor, ktorý obsahuje všetky dáta o stave úlohy. Pre opätovné obnovenie tejto úlohy stačí jednoducho spustiť vytvorený checkpoint, nakoľko je to bežný spustiteľný súbor.

DPVM je upravená verzia PVM 3.3.7, ktorá je prispôsobená tak, aby umožňovala migrovanie úloh. V systéme PVM je totiž na každom uzle v clusteri spustený démon, ktorý zabezpečuje preposielanie a prijímanie

[1] ak by vyšli predpokladané záťaže v algoritme 1 záporné, tak ich zmeníme na 0

[2] GNU C library - súbor knižníc pre jazyk C, ktorý poskytuje základné systémové a iné volania pre systémy UNIX

správ pre PVM úlohy, ktoré sú na tomto uzle. Preto musí vedieť, na ktorom je každá úloha fyzicky umiestnená. DPVM pridáva funkcionalitu, ktorá umožňuje PVM démonom sledovať pohyb migrovaných úloh a korektne aktualizovať informácie o ich lokácii. DPVM taktiež definuje novú funkciu dostupnú pre PVM programy na migrovanie úloh, je to `pvm_move()`. Táto funkcia je využitá aj v monitore, ktorý slúži ako load balancer.

Monitor je aplikácia, ktorá slúži ako centrálny dynamický load balancer pre systém DPVM. Skladá sa z hlavnej úlohy `monitor`, ktorá zbiera informácie o uzloch a úlohách od monitorovacích programov `pvm_slave`. Taktiež sa stará o vyhodnocovanie týchto údajov a prípadnú migráciu vybraných úloh. Samotný monitor sa skladá z piatich threadov. Jeden slúži na zbieranie správ od monitorovacích programov, dva slúžia na vyhodnocovanie a aktualizáciu údajov o uzloch a úlohách v systéme, jeden je spojením na systém PVM a posledný je samotný load balancer, ktorý rozhoduje o vykonávaní migrácií.

Samotný load balancer pracuje tak, že sa snaží zisťovať optimálne rozloženie úloh na uzly pomocou *Branch & Bound* algoritmu, vzhľadom na celkovú záťaž uzlov. Ak sa toto optimálne riešenie líši od aktuálneho, tak vykoná všetky migrácie, ktoré sú na dosiahnutie optimálneho stavu potrebné. Jedinou vecou, ktorá zabraňuje tomu, aby sa nevykonávalo príliš veľa migrácií je to, že pre migráciu pridáva dodatočnú cenu 0.3 záťaže. Tento algoritmus taktiež pri rozhodovaní pracuje výhradne s aktuálnymi údajmi o záťaži uzlov a jednotlivých úloh.

Pre presnejšie informácie o detailnej implementácii systému Dynamite PVM pozri [Iskra00].

2.3.2 Implementácia heuristiky

Moju navrhovanú heuristiku som sa rozhodol implementovať do monitora systému Dynamite PVM, nakoľko už poskytuje infraštruktúru pre monitorovanie stavu počítačového clustera.

Pre efektívne pamätanie si údajov o predošlých spusteniach úloh, ktoré využívam pre výpočet vstupných údajov do algoritmov 1 a 2, som do pôvodného monitora pridal hashovaciu tabuľku, ktorá rieši kolízie zreťazením a ako kľúč používa názov úlohy.

Pre každú úlohu obsahuje nasledovné údaje:

- **task_name** - meno úlohy
- **pct_cpu_avg** - priemerné zaťaženie procesora touto úlohou
- **total_avg** - priemerná dĺžka života tejto úlohy
- **of_writes** - počet zápisov do tejto položky

Takisto som upravil zberač údajov `mon_slave` tak, aby pri ukončení úlohy o tom poslal správu centrálnemu monitoru. Na to, aby včas zistil ukončenie úlohy, využívam štandardnú funkciu PVM `pvm_notify()`, ktorá pri ukončení zadanej úlohy o tom pošle programu správu. Pri zistení, že bola úloha skončená síce stratíme údaje o tom, ako sa správala od posledného zberu údajov[3], ale nakoľko je tento zber vykonávaný každých 20 sekúnd a tomu, že väčšina úloh žije mnohonásobne dlhšie, môžeme túto nepresnosť zanedbať.

Taktiež bolo potrebné na strane centrálneho monitora dorobiť funkcionalitu, ktorá by sa dokázala vyrovnať s tým, ak úlohu zmigruje externý program. Ak sa tak totiž stane, `mon_slave` na zdrojovom uzli migrácie si myslí, že bola úloha ukončená a pošle nám o tom správu. My by sme ju potom chybne zarátali do štatistiky o behu úloh. Preto sa pri každej správe o ukončení úlohy údaje o úlohe najprv vložia do čakacej fronty. Ak sa do 60 sekúnd od ich vloženia objaví úloha na niektorom z iných uzlov, z tejto fronty dáta vyberieme a pripočítame ich k novým aktuálnym údajom o úlohe (ide o priemerné využitie procesora a celkový život úlohy). Ak sú údaje v čakacej fronte viac než 60 sekúnd, až potom je úloha považovaná za ukončenú a jej údaje sú zapísané do hashovacej tabuľky.

Napokon som zmenil samotný load balancer za môj, ktorý pracuje podľa algoritmu 2.

Treba ešte spomenúť, že implementácia bola vykonávaná pre systém *Linux*, nakoľko som k systému *Solaris* nemal prístup, preto je možné, že pre správny chod mojej implementácie na tomto operačnom systéme bude nutné zdrojový kód upraviť.

[3]úloha už totiž v systéme nie je, a preto nemáme ako zistiť, koľko času strávila výpočtami

Kapitola 3

Praktické merania

Pre vyhodnotenie mojej heuristiky som použil merania vykonané na simulačných PVM aplikáciach. Tieto aplikácie som postupne spúšťal najprv so žiadnym load balancerom, potom pôvodným z balíka Dynamite PVM a napokon môjím algoritmom. Taktiež som pri testoch aplikoval externú záťaž, ktorá simulovala zaťaženie vzniknuté činnosťou užívateľov.

3.1 Testovacie aplikácie

3.1.1 Simulačné programy pre PVM

Keďže systém PVM umožňuje aplikáciam spúšťať dcérske úlohy a týmto medzi sebou komunikovať, môžu tým vznikať programy s veľmi komplexnou štruktúrou komunikácie. Nakoľko sa ale moja navrhovaná heuristika nezameriava na komunikáciu, ale skôr na to, na koľko percent tá či oná úloha využíva procesor, navrhol som dve PVM aplikácie, ktoré som pri testoch používal a zameriavajú sa na tento aspekt.

Na sledovanie postupu výpočtu simulačných aplikácií som zaviedol jednotku *sv*, ktorá predstavuje čas potrebný na jedno vykonanie simulačnej funkcie `sim_compute_unit()`. Táto funkcia v skutočnosti vykoná v cykle 60 000 000 multiplikácií[1]. Počet iterácií cyklu som určil experimentálne a na počítačoch, ktoré som používal pri testovaní, trvá tento výpočet pri behu jednej úlohy, ktorá má k dispozícii všetok procesorový čas, cca 0.6 sekundy.

[1]Pri kompilácii tejto funkcie je nutné vypnúť optimalizáciu prekladača *gcc*

Prvá testovacia PVM aplikácia, nazvaná `master1` s dcérskou úlohou `slave1`, je určená na simulovanie CPU intenzívnych aplikácií. `master1` dostane na vstupe počet dcérskych úloh, ktoré má spustiť, počet jednotiek *sv*, ktoré má odovzdať ako jeden balík práce pre `slave1` a nakoniec ešte počet balíkov, ktoré má počas života rozdeliť.

Po spustení teda vytvorí daný počet inštancií `slave1` a postupne im prideľuje balíky úloh. Vždy keď potomok balík dopočíta, pošle otcovi spávu o ukončení a ak ešte neboli všetky balíky rozdané, tak mu otec pridelí ďalší. `slave1` okrem toho ešte vždy, keď vypočíta 100 jednotiek *sv*, pošle o tom otcovi správu. Otec potom počet ohlásených vypočítaných jednotiek *sv* pripočíta k celkovému počtu a vypíše ho spolu s časom na štandardný výstup. Takto je možné sledovať postup výpočtu úlohy pre potreby merania.

Druhá PVM aplikácia je nazvaná `master2` a jej potomok `slave2`. Pracuje rovnako ako `master1` s tým rozdielom, že balíky práce rozdeľuje vždy striedavo polovici úloh. Preto vždy jedna dcérska úloha čaká, kým druhá skončí výpočet.

3.1.2 Simulátor externej záťaže

Pre potrebu simulácie externej záťaže spôsobenej užívateľmi, ktorí nepoužívajú systém PVM som napísal aplikáciu `loadsim`. Táto aplikácia pri spustení načíta vstupný súbor obsahujúci údaje o mieste, čase, dĺžke a veľkosti záťaže, ktorú má vykonať. Prakticky to robí spustením aplikácie `xload` na cieľovom stroji clustera pomocou programu *rsh*. `xload` očakáva na vstupe dve prirodzené čísla označujúce veľkosť záťaže a čas po ktorý ju má vykonávať. Ak je potrebná záťaž väčšia než 1, spustí potrebný počet insštancií samého seba pomocou funkcie `fork()` a potom začne na daný čas počítať v cykle multiplikácie[2].

3.2 Testovacie prostredie

Nakoľko systém Dynamite PVM podporuje len operačné systémy *Linux* a *Solaris*, priklonil som sa pri meraniach k systému *Linux*, s ktorým mám už bohaté skúsenosti.

Spomedzi distribúcií som si vybral *Slackware Linux 9.1*[3]. Jedným

[2] program `xload` je taktiež potrebné kompilovať bez optimalizácie prekladača *gcc*
[3] **`http://www.slackware.com`**

z dôvodov bolo aj to, že najvyššia verzia *GLIBC*, ktorú *dynamite checkpointer* podporuje, je *GLIBC 2.3.2*. Táto je práve použítá v danej distribúcii.

Pri kompilácii systému DPVM vznikli problémy spôsobené vyššou verziou *gcc 3.x*. Tento systém bol vyvíjaný pre *gcc 2.x* a preto bolo potrebné zdrojové kódy pre samotné *DPVM* a *monitor* upraviť[4].

Ďalej som už z predpripraveného systému, kde bol skompilovaný systém DPVM, vytvoril pomocou skriptov *slackware-live*[5] živú CD distribúciu, ktorú som potom používal ako operačný systém pre počítače zapojené do testovania[6]. Toto bolo nutné kôli tomu, že na použité počítače som nemal možnosť inštalovať vlastný operačný systém.

Pre testovanie load balancingu boli použité dva identické clustre po štyroch počítačoch. Všetky tieto počítače obsahovali procesor Intel Celeron 900MHz a 256MB RAM. Spojené boli 100Mbit FastEthernet sieťou. Jednotlivé PVM aplikácie boli zdieľané pomocou NFS. Všetky stroje boli vyhradené čisto pre účely testovania a počas neho na nich nebežali žiadne iné úlohy.

3.3 Postup merania

Pri praktických meraniach som sledoval tri charakteristiky:

- celkový čas behu testovanej PVM aplikácie
- postup výpočtu počas behu PVM programu
- počet vykonaných migrácií

Je jasné, že celkový čas behu aplikácie je hlavným cieľom väčšiny load balancingov, ale aj postup výpočtu môže byť pre niektoré aplikácie dôležitý. Mohli by to byť napríklad aplikácie, ktoré spracúvajú nejaké vizuálne dáta online a potrebujú stabilný prísun spracovaných dát. Z počtu migrácií potom možno usúdiť, ako *múdro* ich systém vykonal. Ak pri tom istom počte migrácií dokáže jeden systém beh aplikácie skrátiť, znamená to, že vykonal migrácie inteligentnejšie než ten druhý.

[4] tieto pozmenené verzie sa nachádzajú na priloženom CD disku

[5] `http://www.linux-live.org`

[6] z priloženého CD disku je možné túto distribúciu spustiť

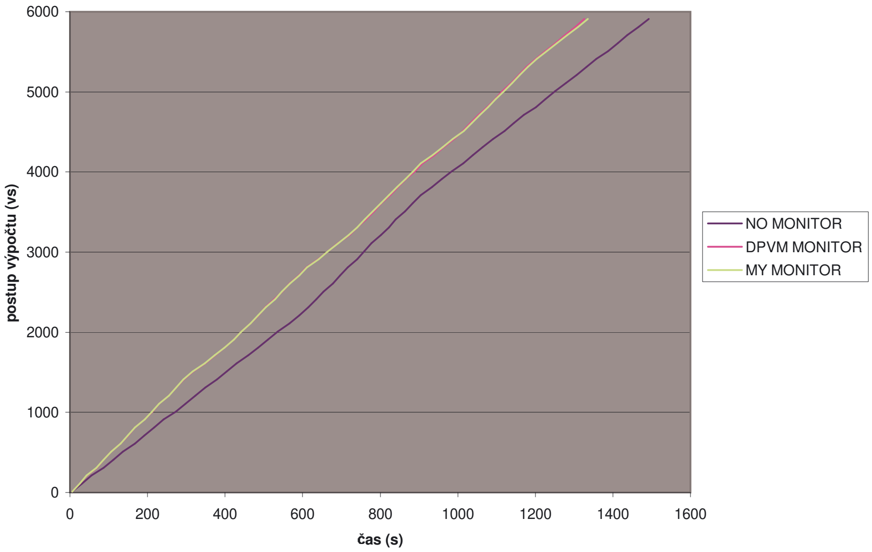
Každý typ aplikácie bol vždy spustený bez load balancingu, s originálnym monitorom systému DPVM a mnou upraveným monitorom.

V prípadoch, kedy je na systém aplikovaná externá záťaž pomocou programu *loadsim*, je to vždy postupne na jeden uzol v clusteri po druhom a vždy na dobu 300 sekúnd so záťažou 2.

Aby boli zachované rovnaké podmienky pri spustení každého merania, tak sa vždy pred spustením testu kompletne vypol systém PVM, zaviedol nanovo a jednotlivé počítače boli do clustera pridávané vždy v tom istom poradí. Taktiež sa vždy merania začínali pri nulovej priemernej záťaži na každom uzli clustera, a ak sa používal pri meraní aj load balancer, bol tento spustený vždy presne 20 sekúnd pred samotným spustením testu.

3.4 Výsledky meraní

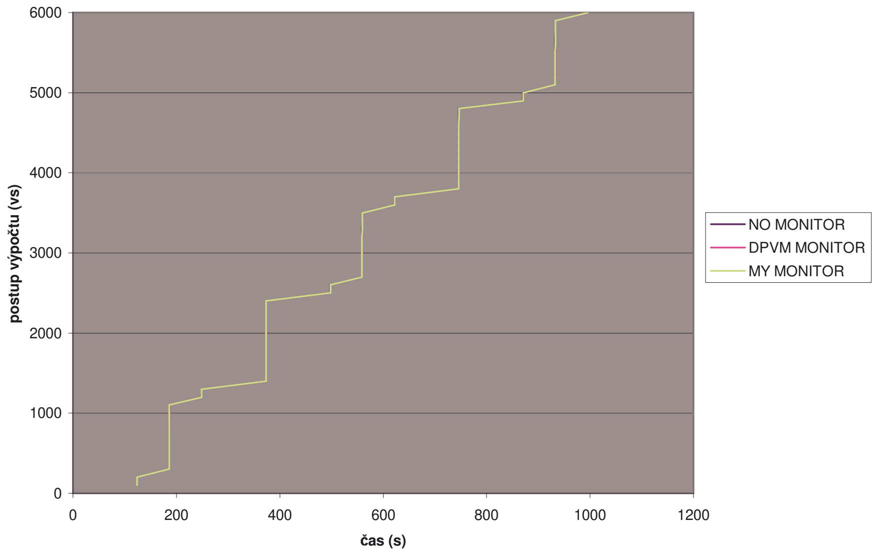
Prvým meraním bolo spustenie programu **`master1 3 100 60`** s **externou záťažou**. Pri tomto teste boli spustené 3 dcérske úlohy `slave1`, ktoré vždy počítali 100 jednotiek *vs* a spolu bolo treba rozdeliť 60 * 100 jednotiek práce, teda spolu 6000. Výsledky merania je možné vidieť na obrázku 3.1.



Obrázok 3.1: **`master1 3 100 60`** + **EXT LOAD**

Keďže sú v clusteri uzly 4 a úlohy sú len 3, vidieť tu ako naozaj migrácia pomáha. Ak je totiž uzol na ktorom beží `slave1` zaťažený externou záťažou, load balancer má možnosť ho zmigrovať na uzol, ktorý nič nerobí. Bez použitia load balancera musí úloha externú záťaž znášať a vidieť to aj na jej spomalení. V tomto prípade si oba load balancery počínali skoro identicky. Oba monitory v tomto prípade skrátili beh úlohy o 11% oproti prípadu bez load balancera a oba vykonali 3 migrácie.

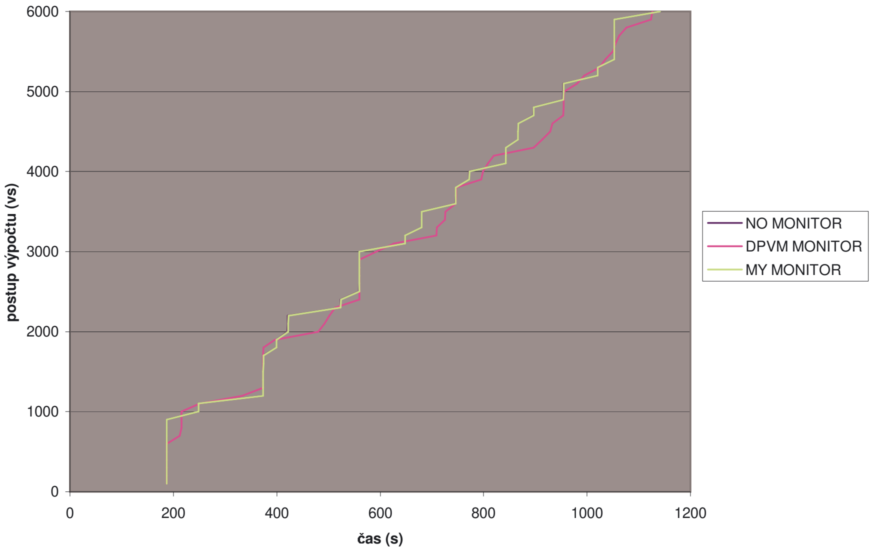
Pri ďalšom meraní som zvýšil počet inštancií `slave1` na 11 a nepoužil externú záťaž. Tento počet úloh viac než dvojnásobne prevyšoval počet uzlov v clusteri a každý uzol bol viac než dvojnásobne preťažený. Graf priebehu výpočtu je možné vidieť na obrázku 3.2.



Obrázok 3.2: **`master1 11 100 60`** + NO LOAD

V tomto meraní sa oba monitory správali rovnako ako systém bez load balancingu. Je to vysvetliteľné tým, že celý systém bol preťažený a navyše boli všetky uzly preťažené viacmenej rovnako, takže sa oba monitory zdržali akýchkoľvek migrácií a preto dosiahli ten istý výsledok ako systém bez load balancera.

Na obrázku 3.4 možno vidieť výsledok merania pre ten istý prípad ako predchádzajúci, teda **`master1 11 100 60`**, ale tentokrát s aplikovaním externej záťaže.

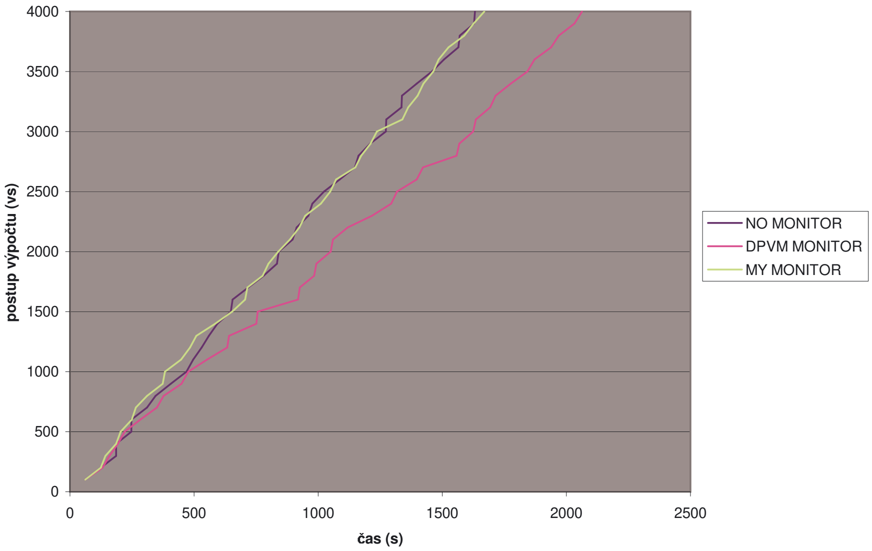


Obrázok 3.3: **`master1 11 100 60`** + **EXT LOAD**

Externá záťaž v tomto prípade trochu zamiešala karty, ale nie príliš. Testovací výpočet v systéme bez load balancera aj s oboma monitormi skončil v rovnakom čase. Zaujímavý je však rozdiel v počte vykonaných migrácií. Kým môj monitor v tomto prípade vykonal migráciu len jednu, pôvodný DPVM monitor ich vykonal až 15. Preto aj môj monitor kopíruje priebeh výpočtu bez load balancera. Vysvetlením, prečo sa overhead z migrácie DPVM monitora neprejavil na dĺžke výpočtu je to, že aj keď bolo urobených veľa migrácií, niektoré z nich boli naozaj prínosom, ale tie ostatné skôr na škodu. Takto sa efekty migrácií prakticky vyrušili. Je to čiastočne vidieť aj z obrázku , kde sa postup výpočtu pri DPVM monitore s postupom pri mojom monitore striedajú v prvenstve. Bolo by to ešte zrejmejšie, ak by sme zvýšili granularitu hlásenia progresu pre `slave1`.

Ako ďalšiu testovaciu aplikáciu som použil program `master2`. Tento už nezaťažuje všetky úlohy naraz, ale vždy len polovicu z nich. Ako prvý som spustil **`master2 4 100 40`** aj s **externou záťažou**. Teda v jednej chvíli vždy bežali dve zo štyroch úloh `slave1` a zostávajúce dve spali. Graf priebehu výpočtu je možné vidieť na obrázku 3.4.

V tomto prípade skončil môj monitor rovnako ako systém bez load balancera, no pôvodný DPVM monitor bol až 20% pozadu. Pôvodný



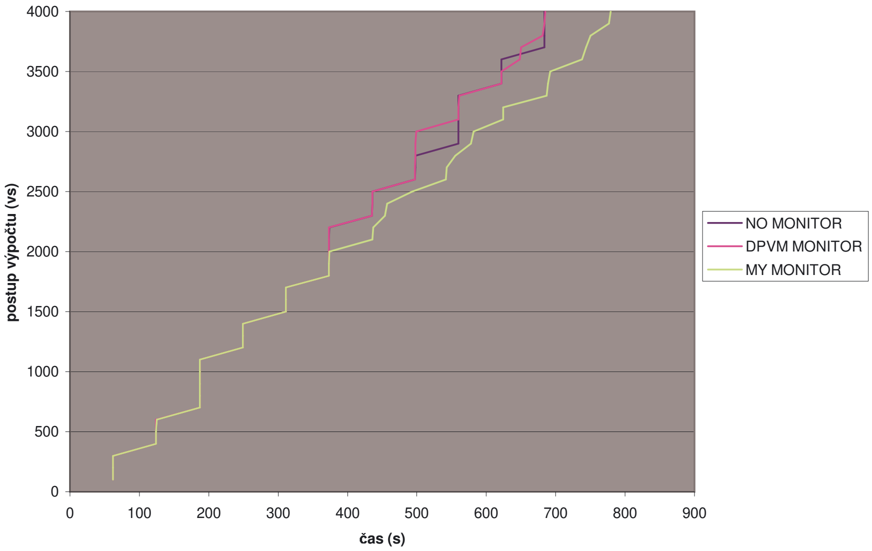
Obrázok 3.4: **`master2 4 100 40`** + EXT LOAD

DPVM monitor tu vykonal 18 migrácií a môj 12. Vysoké počty migrácií sa dajú vysvetliť tým, že monitory boli z relatívne rýchlo meniacich sa úloh zmätené. Jednej úlohe totiž trvá vypočítať 100 *vs* zhruba jednu minútu, nakoľko 1 *vs* trvá na danom systéme asi 0.6 sekundy. Monitorovací systém nestačil dostatočne rýchlo reagovať na zmeny v záťaži uzlov a load balancery preto robili migrácie na základe nie veľmi aktuálnych informácií.

Následne som vykonal test s `master2`, ale bez použitia externej záťaže a zvýšil som počet inštancií `slave2` na 10. Výsledok je na obrázku 3.5.

V tomto prípade skončil môj monitor o 13% horšie než systém bez load balancingu a pôvodný DPMV monitor, ktoré skončili rovnako. Pôvodný DPVM monitor vykonal v tomto teste len 2 migrácie, na rozdiel od môjho, ktorý ich vykonal až 15. Počet migrácií sa odráža aj na celkovom spomalení pri mojom monitore. Opäť sa tu prejavuje pomalšie reagovanie monitorovacieho systému na zmeny v záťaži uzlov.

Ďalej som vykonal rovnaký test ako predchádzajúci, ale s použitím externej záťaže. Teda **`master2 10 100 40`** aj s externou záťažou. Graf merania je uvedený na obrázku 3.6.



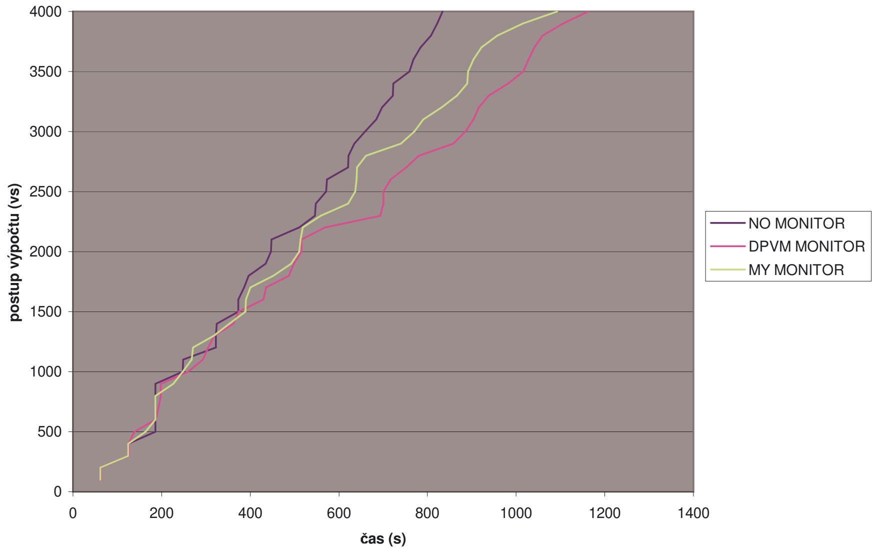
Obrázok 3.5: **`master2 10 100 40`** + NO LOAD

Pri tomto teste dopadli oba monitory horšie než systém bez load balancera. Výpočet pod DPVM monitorom bol oproti systému bez monitora pomalší o 39% a pod mojim monitorom o 31%. Môj systém vykonal 18 migrácií a DPVM monitor až 24. V systéme bežalo z 10 úloh vždy 5 a uzly boli zaťažené podobne, preto migrácie situáciu len zhoršovali. Takéto početné migrovanie sa dá opäť vysvetliť nedostatočnou rýchlosťou reakcie monitoringu na zmeny v systéme.

V ďalšom teste som skombinoval obe testovacie aplikácie. Na začiatku bol spustený **`master1 3 100 100`** a súčasne s ním **`master2 6 100 10`**. *master2* sa spustil celkom štyri krát, prvý na začiatku a ďalší vždy po ukončení predchádzajúceho. Pri tomto teste nebola aplikovaná externá záťaž. Výsledok pre *master1* je uvedený na obrázku 3.7 na strane 34 a graf pre všetky štyri behy *master2* je na obrázku 3.8 na strane 34.

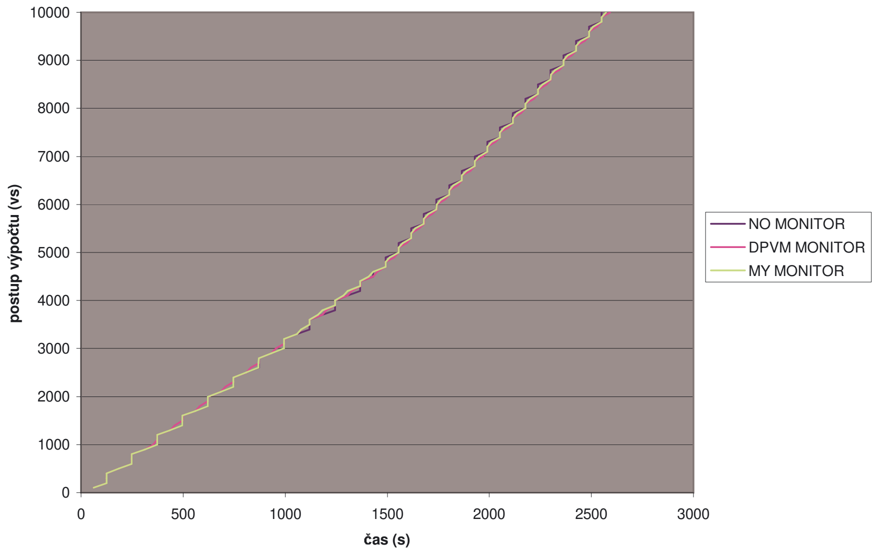
Ako vidieť z obrázkov 3.7 a 3.8, situácia je tu veľmi vyrovnaná. Je to spôsobené tým, že program `master1` využíva len 3 uzly a `master2` má síce 6 synov, ale len traja z nich sú aktívni. Systém preto nie je až tak preťažený a nezostáva veľa miesta pre migrovanie. Oba monitory tu vykonali rovnako po dve migrácie.

V poslednom teste som použil tú istú konfiguráciu ako v predchádzajúcom,

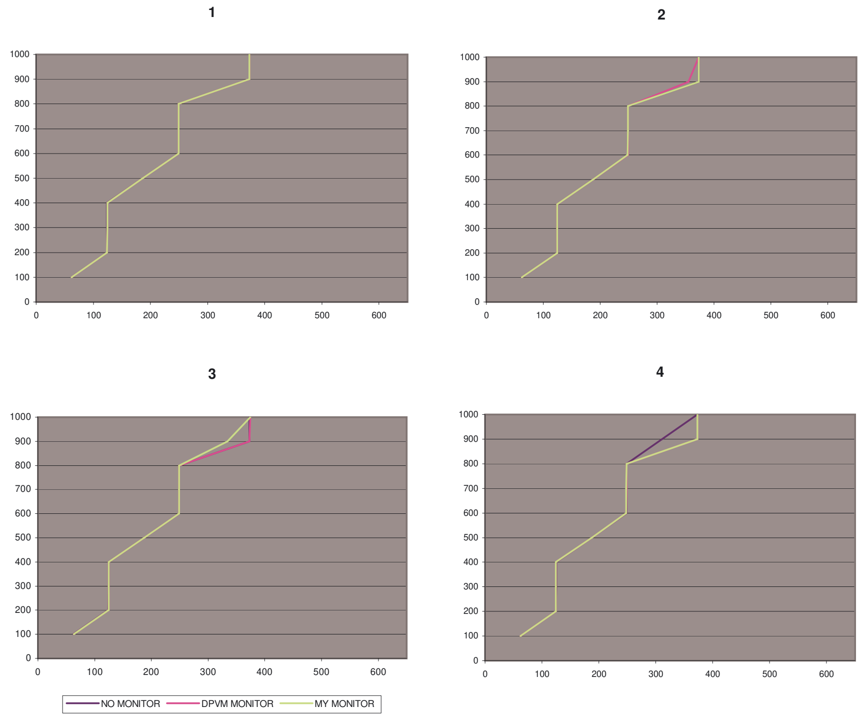


Obrázok 3.6: **`master2 10 100 40`** + NO LOAD

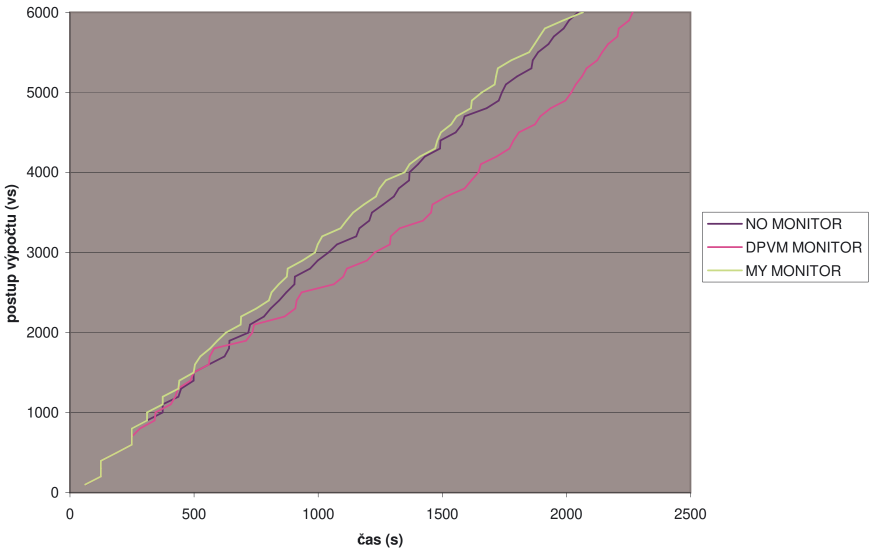
ale pridal som opäť externú záťaž. Výsledky sú uvedené na obrázkoch 3.9 a 3.10 na strane 35.
V poslednom teste sa už čiastočne prejavilo aj využitie histórie behu predchádzajúcich úloh, ktoré je v mojom load balancingu použité. Systém bez load balancingu tu skončil rovnako ako môj systém a pôvodný DPVM monitor za nimi o 10% zaostáva. Z obrázku 3.9 však vidieť, že postup výpočtu bol pri mojej heuristike väčšinu času o niečo vpred. U grafov jednotlivých behov úlohy `master2` to už nie je také markantné, ale to je spôsobené jej relatívne krátkym životom.



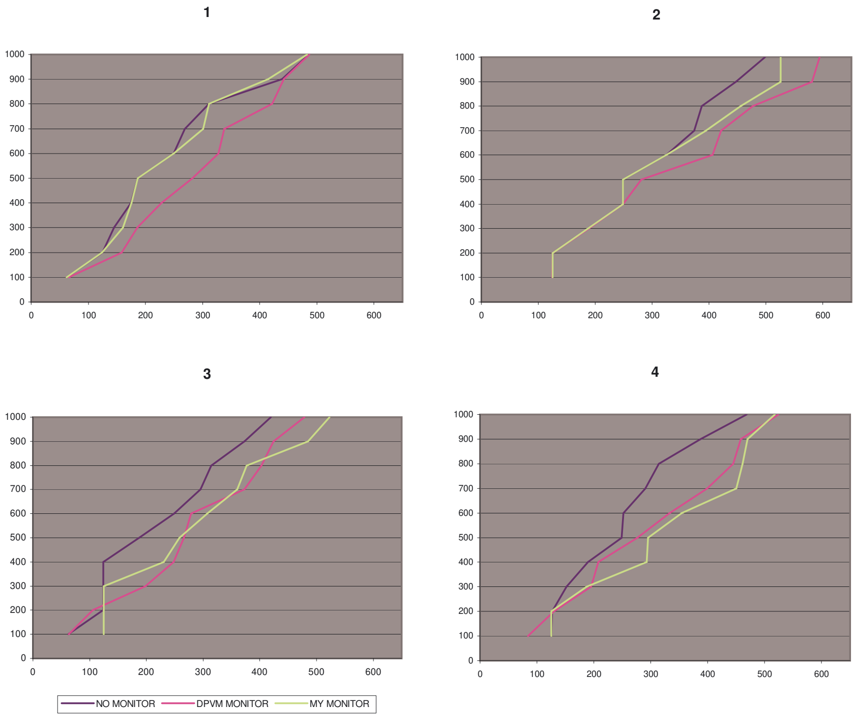
Obrázok 3.7: **`master1 3 100 100`** + **`master2`** + NO LOAD



Obrázok 3.8: 4 * **`master2 6 100 10`** + **`master1`** + NO LOAD



Obrázok 3.9: **`master1 3 100 100`** + **`master2`** + EXT LOAD



Obrázok 3.10: 4 * **`master2 6 100 10`** + **`master1`** + EXT LOAD

Záver

Cieľom tejto práce bolo navrhnúť a experimentálne overiť heuristiku pre dynamický load balancing v systéme PVM, ktorá by využívala migráciu úloh.

Väčšina meraní potvrdila, že je navrhovaná heuristika aspoň tak dobrá ako pôvodný load balancingový algoritmus systému DPVM. Je určite vhodné ju použiť pre vyrovnávanie záťaže v počítačových clusteroch postavených na systéme PVM, v ktorých chceme efektívne využiť nezaťažené stanice. Jej implementácia na operačnom systéme *Linux* a platforme *DPVM* je pomerne jednoducho nasaditeľná v praxi a mohla by tak v mnohých organizáciách pomôcť efektívne zachytiť nevyužitú výpočtovú silu.

Navrhovaná heuristika využíva črty PVM, ktoré sú spoločné pre všetky *message passing* paralelné systémy, a preto je ju možné aplikovať i na ne.

Budúcou prácou by mohlo byť zefektívnenie zberu aktuálnych informácií o uzloch systému a úlohách, ktoré na nich bežia. Prípadne praktická aplikácia tejto heuristiky na iné distribuované systémy.

Literatúra

[Harch97] *Mor Harchol-Balter and Allen B. Downey*: **Exploiting Process Lifetime Distributions for Dynamic Load Balancing**, ACM Transactions on Computer Systems, vol. 15, nr 3 pp. 253-285. August 1997

[Krueg88] *P. Krueger and M. Livny*: **A comparison of preemptive and non-preemptive load distributing**, 8th International Conference on Distributed Computing Systems. IEEE, New York, pp. 123–130 1988

[Over96] *B.J. Overeinder, P.M.A. Sloot, R.N. Heederik and L.O. Hertzberger*: **A Dynamic Load Balancing System for Parallel Cluster Computing**, Future Generation Computer Systems, vol. 12, nr 1 pp. 101-115. May 1996

[Iskra00] *K.A. Iskra, F. van der Linden, Z.W. Hendrikse, B.J. Overeinder, G.D. van Albada and P.M.A. Sloot*: **The implementation of Dynamite - an environment for migrating PVM tasks**, Operating Systems Review, vol. 34, nr 3 pp. 40-55. Association for Computing Machinery, Special Interest Group on Operating Systems, July 2000
`http://www.science.uva.nl/research/scs/Software/dynamite/`

[Geist94] *A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck and V. Sunderam*: **PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing**, MIT Press, Cambridge, Massachusetts, 1994
`http://www.epm.ornl.gov/pvm/`

[Ju95] *Jiubin Ju, Gaochao Xu and Kun Yang*: **An intelligent dynamic load balancer for workstation clusters**, SIGOPS Oper. Syst. Rev., vol. 29, nr 1 pp. 7-16. 1995

[Miloj00] *D. Milojicic, F. Douglis, F., Y. Paindaveine, R. Wheeler, and S. Zhou*: **Process Migration Survey**, ACM Computing Surveys, vol. 32, nr 3 pp. 241-299, September 2000

[Kunz91] *T. Kunz*: **The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme**, IEEE Transactions on Software Engineering, vol. 17, nr 7 pp. 725–730. 1991

[Plank95] *J. S. Plank, M. Beck, G. Kingsley and K. Li*: **Libckpt: Transparent Checkpointing under Unix**, Usenix Winter Technical Conference, New Orleans, LA, pp. 213-223, 1995 **`http://www.cs.utk.edu/~plank/plank/papers/papers.html`**

[Kohl98] *J. A. Kohl, P. M. Papadopoulos*: **Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS**, 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, Welches, OR, August 1998

[Barak98] *A. Barak and O. La'adan*: **The MOSIX Multicomputer Operating System for High Performance Cluster Computing**. Journal of Future Generation Computer Systems (13) 4-5, pp. 361-372, March 1998 **`http://www.mosix.org/mosultp/mosixhpcc.pdf`**

[Oust88] *J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch*: **The Sprite Network Operating System**, IEEE Computer, pp. 23–26, 1988

[Plank95a] *J. S. Plank, J. Xu and R. H. B. Netzer*: **Compressed Differences: An Algorithm for Fast Incremental Checkpointing**, University of Tennessee, CS-95-302, August 1995 ***http://www.cs.utk.edu/~plank/plank/papers/CS-95-302.html***

[Cher88] *D. R. Cheriton*: T**he V Distributed System**, Communications of the ACM vol. 31, nr 3 pp. 314–333, 1988

[Marei02] *S. Marei, Al-Amri and Rana Ejaz Ahmed*: **New job selection and location policies for load-distributing algorithms**, Int. J. Netw. Manag., John Wiley & Sons, Inc., New York, NY, USA, vol. 12, nr 3 pp. 165-178

Zoznam príloh

CD-ROM obsahujúci:

- zdrojový kód implementácie navrhovanej heuristiky
- zdrojový kód použitých testovacích aplikácií
- zdrojový kód *DPVM* a *DPVM monitora* upravený pre *gcc 3.x*
- predkompilovaný *dynamite checkpointer* pre *gcc 2.3.2*
- live distribúciu *linuxu* s predkompilovaným systémom DPVM použitú pri testovaní