UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

VYROVNÁVANIE ZAŤAŽENIA NA NESPOĽAHLIVÝCH PROCESOROCH

(Diplomová práca)

JÁN SLIACKY

Školiteľ: Dr. Tomáš Plachetka

Bratislava, 2010

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry.

V Bratislave, 7. mája 2010 .....

Ján Sliacky

Ďakujem svojmu diplomovému vedúcemu Dr. Tomášovi Plachetkovi za cenné rady a usmernenia, rodine za podporu a tiež priateľom za pochopenie pri písaní tejto práce.

Abstrakt

Práca sa zaoberá návrhom a implementáciou algoritmu dynamického distribuovaného vyrovnávania zaťaženia na nespoľahlivých procesoroch, to znamená takých, u ktorých môže kedykoľvek a na akúkoľvek dlhú dobu nastať chyba zrútením. Ďalej obsahuje praktické merania navrhnutého modelu škálované viacerými veličinami a zhodnotenie efektivity paralelného výpočtu navrhnutého modelu.

Kľúčové slová: vyrovnávanie zaťaženia, spoľahlivosť, odolnosť voči chybe

Obsah

Kapitola 1

Úvod

1.1 Motivácia

Dual-channel RAM, Quad-core CPU, Multi-GPU, RAID... Trend posledných rokov smeruje ku zvyšovaniu výkonu komponentov ich násobením, taktiež veľký rozmach v posledných rokoch zaznamenal grid computing, známe projekty ako SETI@Home, FOLDING@Home... V mnohých vedeckých oblastiach ako fyzika a biológia sa vykonávajú výpočty pozostávajúce z veľkého množstva nezávislých úloh. Tým vzniká aj potreba rozumného rozdeľovania úloh (vyrovnávania záťaže), aby sme výpočtový potenciál využili čo najefektívnejšie.

Okrem toho „klasické"(von Neumann-ovského typu) počítače ako ich poznáme dnes, narazia o pár rokov na fyzikálne hranice. Budú postupne nahrádzané novými, ako sú kvantové či biologické počítače, ktoré ešte nie sú v tom štádiu vývoja, aby mohli „klasické" počítače nahradit' už dnes. Biologické počítače sú pomerne výkonné, no zatiaľ vysoko nespoľahlivé.

Spoľahlivost' súčasných počítačov je dnes veľmi dobrá, aj keď nie stopercentná. Vo väčšine prípadov to postačuje, ale sú niektoré oblasti použitia počítačov, kde sú chyby neprípustné, mohlo by to spôsobit' obrovské škody (letectvo, zdravotníctvo), alebo prípadná oprava chýb nie je možná (vo vesmíre). Aj keď je výskyt chyby v dnešných počítačoch veľmi ojedinelý, nemôžeme vznik takejto chyby zanedbat' a to najmä v prípade, že čas výpočtu trvá rádovo mesiace, prípadne úlohu riešia naraz tisíce počítačov. Bez mechanizmov odolnosti voči chybám by to mohlo znamenat' až stratu celého výpočtu.

1.2 Cieľ diplomovej práce

Cieľom diplomovej práce je navrhnúť model vyrovnávania zaťaženia na sieti pozostávajúcej len z nespoľahlivých procesorov (t.j. takých, ktoré môžu zlyhať kedykoľvek a na ľubovoľne dlhú dobu), následne ho implementovať a škálovaním jeho atribútov pri experimentoch zhodnotiť jeho efektivitu paralelného výpočtu.

1.3 Členenie diplomovej práce

Druhá kapitola čitateľa oboznamuje so základnými pojmami v oblasti vyrovnávania zaťaženia a spoľahlivosti. Tretia kapitola pojednáva o samotnom návrhu modelu vyrovnávania zaťaženia na nespoľahlivých procesoroch. V štvrtej je popis implementácie modelu. V piatej kapitole sú popísané postupy jednotlivých experimentov spolu s meraniami, ktoré sú v šiestej kapitole zhrnuté, je z nich vyvodený záver a navrhnuté ďalšie možnosti rozšírenia modelu.

Kapitola 2

Prehľad problematiky

2.1 Základné pojmy a označenia

2.1.1 Terminológia

Definícia 1 ***Distribuovaný systém*** *sa skladá zo súboru výpočtových entít*[1]*, pričom tieto entity musia mať možnosť spolu komunikovať a každá z nich môže pracovať aj samostatne. Užívatelia pritom distribuovaný systém vnímajú ako samostatnú výpočtovú entitu. Výpočtové entity sa zvyčajne označujú ako* ***uzly*** *distribuovaného systému.*

Definícia 2 ***Paralelné počítanie*** *je súčasné počítanie jednej úlohy na viacerých výpočtových entitách za účelom zrýchlenia výpočtu.*

Definícia 3 ***Podúloha*** *(task, job) s časovou zložitosťou T je činnosť, ktorej výpočet trvá procesoru čas T.*

Definícia 4 ***Úloha*** *(work) s časovou zložitosťou T je množina podúloh, ktorých súčet časových zložitostí je rovný T.*

2.1.2 Vyrovnávanie zaťaženia

Definícia 5 ***Vyrovnávanie zaťaženia*** *(Load-balancing) je technika používaná na rozloženie záťaže medzi dva a viac počítačov, sieťových liniek, diskov alebo iných zdrojov, za*

[1] V nasledujúcich častiach je namiesto pojmu výpočtová entita použitý pojem procesor, teda v nasledujúcom kontexte sa neberie ako skutočný procesor (CPU), ale ako výpočtová entita, ktorá s inou výpočtovou jednotkou nemusí nič zdieľať.

účelom získania optimálneho zaťaženia týchto zdrojov, zvýšenia výkonu alebo zrýchlenia výpočtu.

V distribuovaných systémoch často vzniká nevyrovnanosť zaťaženia jednotlivých komponentov. Príčiny bývajú rôzne. Najčastejšie sa stretávame s problémom, že systém býva používaný aj na iné ako paralelné výpočty. Tiež sa často stáva, že nejaké úlohy čakajú na výsledok od iných úloh. Keď sa v systéme objavia viaceré takéto úlohy, niektoré uzly sa stávajú preťaženými. Nevyrovnanosť zaťaženia jednotlivých uzlov vzniká tiež vtedy, keď sa uzly nachádzajú v prostredí heterogénneho distribuovaného systému (nie všetky uzly sú rovnaké).

Nasledujúce atribúty priamo ovplyvňujú výber prístupu vyrovnávania zaťaženia.

Pre *prostredie* distribuovaného systému sú podstatné dva atribúty:

- *Homogenita* - popisuje podobnosť jednotlivých procesorov v distribuovanom systéme, teda či majú podobnú výpočtovú silu.
- *Umiestnenie* - na ktorom procesore je podúloha riešená. Nie vždy majú procesory rovnakú funkčnú špecifikáciu a niektoré špecifické úlohy môžu vyžadovať ich riešenie na konkrétnom procesore. Niekedy je tiež vhodné určitú množinu podúloh riešiť na rovnakom procesore, aby sme sa vyhli nadbytočnej komunikácii a distribúcii medzivýsledkov.

Pre *úlohu* sú podstatné atribúty jednotlivých podúloh, z ktorých pozostáva a to:

- *Veľkosť podúloh* - v našom prípade ich dĺžka trvania. Ak sú veľkosti podúloh rádovo rozdielne a je použitý nesprávny prístup k vyrovnávaniu zaťaženia, môže vzniknúť pomerne veľká nevyrovnanosť zaťaženia jednotlivých procesorov.
- *Závislosť podúloh* - reprezentuje nutnosť vykonávať podúlohy v určitom poradí. Spracovanie jednej môže totiž závisieť na výsledku inej.

2.2 Prístupy pre vyrovnávanie zaťaženia

Prístupy k vyrovnávaniu zaťaženia môžeme rozdeliť podľa toho, kedy získame atribúty popísané v sekcii 2.1.2: [Zam05]

- *Statický prístup* - stav všetkých atribútov je dostupný ešte pred začatím výpočtu. V praxi použiteľný len tam, kde sa záťaž mení minimálne, prípadne vieme záťaž predpovedať. Jeho výhodou je, že nemá žiadnu nadbytočnú réžiu na zisťovanie informácií o systéme počas behu. Skoro nepoužiteľný sa stáva v systéme, kde sa dynamicky mení záťaž na jednotlivých procesoroch. [TT85]

- *Polostatický prístup* - stav atribútov nie je dostupný na začiatku spracovania celej úlohy, ale na začiatku každého samostatného celku úlohy, prípadne iného dobre definovaného bodu. Používa sa v systémoch, kde sa záťaž mení veľmi pomaly a kde je dôležité *umiestnenie* podúloh.

- *Dynamický prístup* - stav atribútov je dostupný až počas výpočtu alebo po ňom. Atribúty sú aktualizované počas behu a podľa takto získaných dát sa prideľujú zvyšné podúlohy. Tento prístup je schopný omnoho dynamickejšie reagovať na zmeny záťaže na jednotlivých procesoroch. Avšak daňou za túto pružnosť, na rozdiel od statického prístupu, je réžia pri získavaní stavu atribútov.

Prístupy k vyrovnávaniu zaťaženia môžeme tiež rozdeliť z pohľadu, či vyžadujú centrálny procesor na prideľovanie podúloh jednotlivým procesorom, na:

- *Centralizované vyrovnávanie zaťaženia* (centralized load balancing, centralized scheduling, self-scheduling) - Pri tomto prístupe všetky podúlohy prideľuje každému procesoru centrálny procesor. Pritom procesory si nemôžu medzi sebou posielať podúlohy, čiže čo procesor dostane, to musí vyriešiť. Výhodou v tomto prístupe je, že nevzniká nadbytočná réžia premiestňovaním podúloh. Podúloha sa vždy prenesie iba raz na cieľový procesor. Aby bol tento prístup efektívny, musíme ale poznať časovú zložitosť podúloh. Ak tieto znalosti nemáme, môže byť tento prístup značne

neefektívny, aj napriek tomu, že existujú rôzne dobré heuristiky na prideľovanie podúloh. V priemernom prípade sú tieto metódy ale pomerne efektívne a sú vo väčšine prípadov aj použité. Ďalšou nevýhodou je *nutnosť* existencie centrálneho procesora, ktorý, keďže riadi celé vyrovnávanie zaťaženia, musí byť aj stopercentne spoľahlivý. Typický zástupcovia tohto prístupu sú fixed-size chunking, guided self-scheduling, tapering, factoring [Hag97, Pla03]

- *Distribuované vyrovnávanie zaťaženia* (Work-stealing, Work crews) - Základom je presúvanie úloh medzi procesormi, teda pri tomto prístupe získavame možnosť riešiť problém preťaženého procesora. Nevýhodou je réžia, ktorá vzniká pri procese presúvania podúloh. Pri tomto type vyrovnávania zaťaženia sa budeme musieť rozhodnúť podľa povahy prostredia a charakteru podúloh, ktoré možnosti pri nasledovných výberoch zvoliť. [vNKB01, BJK+96]
  - *Výber iniciátora*
    - *Iniciované odosielateľom* (sender-initiated) - Preťažený procesor sa snaží presunúť nejaké podúlohy na menej vyťažené procesory. Používa sa v systéme, kde je málo preťažených procesorov.
    - *Iniciované prijímateľom* (receiver-initiated) - Voľný procesor sa snaží získať podúlohy od zaťaženého procesora. Používa sa v systéme, kde je naopak málo voľných procesorov.
    - *Symetrické* - Spája výhody predošlých dvoch prístupov.
  - *Výber zdroja/cieľa presunu*
    - *Asynchrónny round robin* - Procesor kontaktuje o podúlohy i-ty procesor, pričom i je lokálna premenná (teda pre každý procesor vlastná) a po každej žiadosti nasleduje $i := (i + 1)\ mod\ procs$.
    - *Globálny round robin* - Procesor kontaktuje o podúlohy i-ty procesor, pričom i je globálna premenná (zdieľaná všetkými procesormi) a po každej žiadosti nasleduje $i := (i + 1)\ mod\ procs$.
    - *Náhodne* - Procesor kontaktuje o podúlohy náhodný procesor.

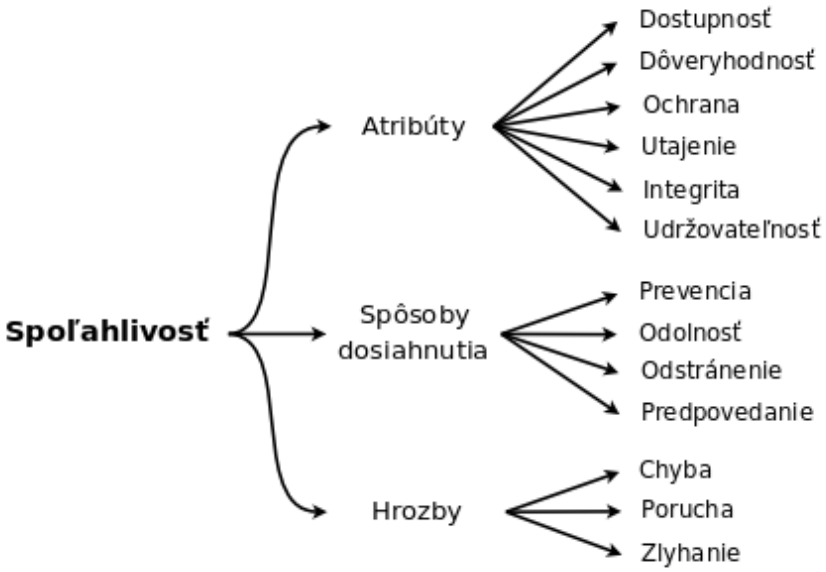
    - *Všetkým* - Procesor kontaktuje o podúlohy všetky procesory nachádzajúce sa v sieti.

– *Výber podúloh* - V prípade, že sú medzi podúlohami závislosti, musia sa vyberať také podúlohy, aby procesor, ktorému ich posielame, mohol hneď začať počítať. Tiež treba zohľadniť umiestnenie jednotlivých podúloh.

2.3 Spoľahlivosť

Definícia 6 ***Spoľahlivosť*** *(dependability) počítačového systému je schopnosť poskytnúť službu, na ktorú bol navrhnutý a ktorej možno preukázateľne dôverovať.*

Systematické vysvetlenie pojmu spoľahlivosť pozostáva z troch častí: atribúty spoľahlivosti, hrozby, ktoré ovplyvňujú spoľahlivosť a spôsoby na dosiahnutie spoľahlivého systému. (obrázok 2.1) [LAK92, ALR01]



Obr. 2.1: spoľahlivosť

2.3.1 Atribúty spoľahlivosti

Nasledujúce atribúty určujú ako je systém spoľahlivý:

- *Dostupnosť* (availability) - pripravenosť systému poskytovať správnu službu
- *Dôveryhodnosť* (reliability) - kontinuita poskytovanej služby
- *Ochrana* (safety) - absencia katastrofických dopadov na užívateľov a prostredie
- *Utajenie* (confidentiality) - všetky prístupy k informáciám sú autorizované
- *Integrita* (integrity) - systém sa nikdy nedostane do nesprávneho stavu
- *Udržovateľnosť* (maintainability) - schopnosť za behu opravovať a modifikovať systém

2.3.2 Hrozby

Definícia 7 ***Hrozba*** *(threat) je okolnosť, ktorá môže viesť k zníženiu miery spoľahlivosti systému*

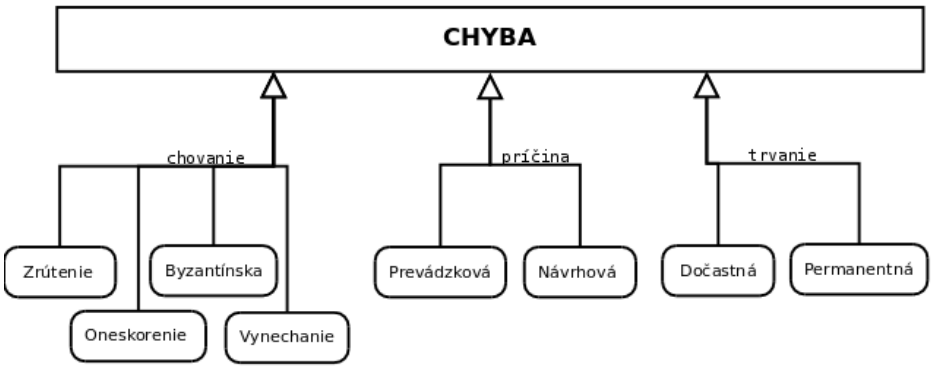
Hrozby podľa dopadu na systém rozdeľujeme na:

- *Zlyhanie* (failure) - nastáva, keď sa práve bežiaci systém vychyľuje z funkčnej špecifikácie. Zlyhanie býva spôsobené poruchou.
- *Porucha* (error) - je reprezentovaná ako nesprávny stav systému, ktorý nie je povolený funkčnou špecifikáciou systému. Porucha býva zapríčinená nejakou chybou.
- *Chyba* (fault) - je vadou v systéme. Môže, ale nemusí viesť k poruche. Tiež ale môže viesť k niekoľkým poruchám súčasne.

Klasifikácia chýb

Chyby rozdeľujeme podľa nasledujúcich kritérií (obrázok 2.2)

- *Trvanie výpadku*
  - *Permanentné* (permanent) - permanentná ostáva v systéme, pokým ju niekto neodstráni.
  - *Dočasné* (transient) - dočasná chyba môže zmiznúť kedykoľvek bez nejakého zásahu. Z hľadiska závažnosti je dočasná chyba vážnejšia, pretože je ťažšie diagnostikovateľná, a keď zmizne predtým, ako ju odstránime a zabezpečíme sa proti jej výskytu, nevieme kedy môže nastať znovu.
- *Príčina vzniku chyby*
  - *prevádzkové* (operational) - prevádzkové chyby sa, na rozdiel od permanentných, vyskytnú až počas prevádzky systému. Väčšinou sú spôsobené hardvérovými chybami (zlyhanie procesora, disku...).
  - *návrhové* (design) - návrhové chyby sú dôsledkom chýb, ktoré vznikli v dobe návrhu systému. Dôsledne navrhnutý systém by mal ale všetky chyby eliminovať pomocou prevencie, ale v praxi to nie je zvyčajne reálne. Z týchto dôvodov je väčšina systémov odolných voči chybám postavených na predpoklade, že návrhové chyby sú v návrhu, ale mechanizmy tretích strán budú systém pred nimi ochraňovať.
- *Správanie systému pri výskyte chyby*
  - *vynechaním* - systém zlyhá pri poskytovaní služby
  - *oneskorením* - systém nestihne dokončiť službu načas
  - *zrútením* - systém úplne skončí a neposkytuje naďalej žiadne služby
  - *byzantínske* - sú chyby vzniknuté ľubovoľným chovaním (ľubovoľnou kombináciou predchádzajúcich)



Obr. 2.2: klasifikácia chýb

2.3.3 Spôsoby na dosiahnutie spoľahlivosti

Na dosiahnutie spoľahlivosti v počítačových systémoch sa používajú štyri základné techniky:

- *Prevencia voči chybe* (fault prevention) - zabraňuje výskytu chýb
- *Odolnosť voči chybe* (fault tolerance) - zabezpečuje poskytovanie služby v prípade výskytu chyby
- *Odstránenie chyby* (fault removal) - rieši ako chyby odstrániť, prípadne znížit' ich dopad
- *Predpovedanie chyby* (fault forecasting) - predpovedá výskyt potencionálnych chýb

Prevencia voči chybe

Úlohou *prevencie voči chybe* je zabezpečit' obmedzenie množstva príčin vzniku chýb. To sa docieľuje mnohými kontrolnými technikami kvality počas návrhu a výroby. Sem možno zaradit' viaceré programátorské techniky ako štruktúrované programovanie, modulárne programovanie, objektovo-orientované programovanie, používanie návrhových vzorov, test-driven development... Taktiež medzi kontrolné techniky patria firewally, antivírusové programy a pod.

Odolnosť voči chybe

Definícia 8 ***Odolnosť voči chybe*** *(Fault-tolerance) je vlastnosť, ktorá umožňuje systému poskytovať služby aj po zlyhaní (jedného alebo viacerých) komponentov.*

Obr. 2.3: procedúra odolnosti voči chybe

Typická procedúra odolnosti voči chybe pozostáva zo štyroch krokov (obrázok 2.3). *Detekcia poruchy* je proces identifikovania, že systém je v neplatnom stave. To znamená, že nejaký komponent v systéme zlyhal. Nasleduje fáza *izolácie poruchy*, aby sa dopad poruchy nešíril ďalej v systéme. Vo fáze *zotavenia sa z poruchy* je porucha a hlavne jej efekt odstránená obnovením systému do platného stavu. Nakoniec vo fáze *ošetrenia chyby* hľadáme chybu, ktorá spôsobila poruchu a snažíme sa ju izolovať. Pri tejto procedúre sa najprv riešia dopady poruchy a až nakoniec dôvody jej vzniku. Toto poradie krokov je vhodnejšie, pretože fáza ošetrenia chyby môže trvať pomerne dlho, keďže jedna chyba typicky spôsobí kaskádovito niekoľko porúch. [JGKR05, RW02, Jeo96]

- *Detekcia poruchy* (error detection) Najčastejšie sa stretávame s nasledujúcimi technikami na detekciu poruchy:

  - *Kontrola replikáciou* (Replication checks) - niekde označovaná aj kontrola redundanciou, je kontrola, pri ktorej je služba vykonávaná súčasne na viacerých komponentoch (priestorová redundancia), alebo je na jednom vykonaná viackrát po sebe (časová redundancia). Tieto výsledky sa vzájomne porovnajú a buď budeme od každého očakávať rovnaký výsledok, alebo nám bude stačiť rovnaký výsledok od väčšiny, inak indikujeme poruchu daných komponentov.[Gar99]
  - *Kontrola oneskorením* (Timing checks) - je používaná na zistenie chýb oneskorením. Typicky je spustený nejaký časovač s maximálnym časom, ktorý má služba na dokončenie. Ak časovač vyprší pred dokončením služby, je indikovaná chyba oneskorením.
  - *Kontrola obmedzením* (Run-time constraints checking) - kontrolujú sa isté obmedzenia, ako napríklad hraničné hodnoty premennej, ktoré sa nemôžu prekročiť počas behu. Toto má však za následok značnú réžiu naviac. Tiež sa zvyknú porovnávať celé štruktúry metódou porovnávania kontrolných súčtov.
  - *Diagnostická kontrola* (Diagnostic checks) - slúži na zistenie, či komponent funguje korektne. Najčastejšie sa dávajú komponentu konkrétne vstupy a získaný výsledok sa porovnáva s očakávaným výsledkom.

- *Izolácia poruchy* (damage confinement) - pri tejto technike sa zisťuje rozsah poruchy, teda ktoré časti systému sú ovplynené poruchou, a izolovať ich. K tomu potrebuje poznať procesy v systéme, prehľadávať ich a stanoviť hranice, čo je potrebné izolovať a čo nie.

- *Zotavenie sa z poruchy* (error recovery) - je proces, pri ktorom sa systém musí dostať do platného stavu. K tomu sa používajú nasledujúce prístupy:

  - *Spätné zotavenie* (backward error recovery) - Systém je obnovený do jedného z predošlých platných stavov. Uloženie tohto stavu sa vykonáva buď metódou checkpointingu alebo metódou loggingu. Pri checkpointingu sa ukladá priebežne globálny stav systému. Pri loggingu sa ukladajú správy, ktoré sú vykonávané a po chybe sa systém

obnoví do predošlého platného stavu a znova odpovedá na správy v poradí, v akom mu prišli.

- *Dopredné zotavenie* (forward error recovery) - Systém sa snaží dostať z neplatného stavu do nového platného stavu. To je možné až po dobrej znalosti chyby a jej následnej izolácii.

- *Ošetrenie chyby* (fault treatment) - chyba sa najprv izoluje a potom opraví. Procedúra akou bude chyba opravená závisí na type chyby. Permanentné vyžadujú nahradenie chybného komponentu novým. To vyžaduje mať nejaký záložný komponent, ktorý v prípade poruchy komponentu bude dočasne slúžiť na obnovenie systému a nahradenie funkcionality chybného komponentu.

Odstránenie chyby

Odstránenie chyby je vykonávané aj vo fáze vývoja, aj vo fáze prevádzky. Vo fáze vývoja pozostáva životný cyklus systému z troch krokov: *verifikácia*, *diagnostika* a *korekcia*. Verifikácia je proces kontroly, či systém spĺňa dané vlastnosti, nazývané aj verifikačné podmienky. Ak nie, nasledujú ďalšie kroky, a to diagnostika chyby, ktorá zabránila splneniu verifikačnej podmienky a jej následná korekcia. Vo fáze prevádzky sa delí odstraňovanie chýb na korekčnú opravu a prevenčnú ochranu. Korekčná je zameraná na odstraňovanie chýb, ktoré už spôsobili nejakú poruchu a boli zachytené. Prevenčná oprava sa snaží odstrániť chybu ešte predtým ako spôsobí poruchu počas prevádzky.

Predpovedanie chyby

Predpovedanie chyby vyhodnocuje chovanie systému s ohľadom na výskyt možnej chyby. Toto vyhodnocovanie má dva aspekty:

- *kvalitatívny* - zameriava sa na identifikáciu, klasifikáciu, ohodnocovanie typov chýb, ktoré môžu viesť ku zlyhaniu systému.

- *kvantitatívny* - zameriava sa na vyhodnocovanie pravdepodobností, ako výskyt chyby ovplyvní atribúty spoľahlivosti.

Kapitola 3

Návrh modelu vyrovnávania zaťaženia na nespoľahlivých procesoroch

3.1 Motivácia

Motiváciou pre návrh tohto modelu bola predstava čo najrobustnejšieho (v rámci výskytu chýb zrútením) riešenia vyrovnávania zaťaženia v sieti nespoľahlivých procesorov s akceptovateľným dopadom na výkon. Preto, ako sa ďalej dočítate, model nemá žiadne zásadné obmedzenie, ktoré by sa určitými úpravami nedalo zrealizovať.

3.2 Základné pojmy a označenia

Definícia 9 ***Nespoľahlivý procesor*** *je procesor, u ktorého je predpoklad, že kedykoľvek môže nastať zlyhanie pádom (crash failure).*

Definícia 10 ***Sieť nespoľahlivých procesorov*** *je množina pozostávajúca len z nespoľahlivých procesorov, pričom sú medzi sebou prepojené tak, že ľubovoľné dva procesory môžu medzi sebou komunikovať.*

3.3 Čo nepredpokladá model

Snahou pri návrhu modelu bolo čo najmenšie obmedzenie, aby sa dal použiť v čo najväčšom množstve prostredí. Preto boli na začiatku určené predpoklady, ktoré nemôžu model obmedzovať, a z nich sa potom odvíjal samotný návrh modelu. Model teda nepredpokladá:

1. **Spoľahlivosť aspoň jedného procesora**
   - Žiadny z procesorov nemusí byť spoľahlivý, čiže nemožno sa spoliehať na centrálnu autoritu, ktorá bude celý proces riadiť.
2. **Znalosť časovej zložitosti podúloh**
   - Časové zložitosti podúloh budú známe až po ich spracovaní.
3. **Aspoň jeden procesor pracujúci v každom momente**
   - V jednom momente môžu zlyhať aj všetky aktívne procesory.

3.4 Čo predpokladá model

Všetky nasledujúce predpoklady slúžia len na zjednodušenie modelu, ale pritom budú stále zabezpečené podmienky zo sekcie 3.3.

1. **Sieť nespoľahlivých procesorov tvorí kompletný graf**
   - Navrhnutý model nezaujíma ako je riešená nižšia vrstva topológie siete. To, že sa pre model tvári ako kompletný graf, treba zabezpečiť na tejto nižšej vrstve.
2. **Počet procesorov je vopred známy**
   - Počet procesorov by sa mohol aj meniť, len treba zabezpečiť po pripojení procesora distribúciu celej úlohy novo pripojenému procesoru.
3. **Linky medzi procesormi sú spoľahlivé**

- Rovnako ako prvý predpoklad. Spoľahlivosť liniek musí byť riešená na nižšej vrstve.

4. **Úloha sa zmestí každému procesoru do pamäte**

   - Každý procesor musí byť schopný dopočítať úlohu, aj keď všetky ostatné procesory zlyhajú, teda má všetky podúlohy vo svojej lokálnej pamäti. Tento predpoklad je ale riešiteľný na hardvérovej úrovni.

5. **Počet podúloh je vopred známy a nemení sa**

   - Ak by sa počet podúloh menil, bolo by potrebné zabezpečiť korektnú distribúciu všetkých nových podúloh ostatným procesorom.

6. **Podúlohy sú navzájom nezávislé**

   - Tento predpoklad by sa dal zovšeobecniť na predošlý predpoklad, a to, že podúlohy, ktoré od niečoho závisia, sa budú do systému pridávať postupne, ako budú splňované ich závislosti.

7. **Každý procesor má na začiatku celú úlohu vo svojej pamäti**

   - Tiež špeciálny prípad pridania nového procesora do siete. Opäť sa musí zabezpečiť korektné rozdistribuovanie úlohy.

8. **Nespoľahlivý procesor po zlyhaní nestráca údaje z pamäti**

   - Procesoru zostane celá úloha a aj stav podúloh. Keby ich strácal, tak by to bol tiež len špeciálny prípad pridania nového procesora.

Čiže tieto predpoklady sú buď zjednodušením z praktických dôvodov (2., 5., 6., 7. a 8. predpoklad), alebo sú obmedzenia (najmä hardvérové), ktoré sú riešiteľné na nižších úrovniach ako je navrhovaný model (1., 3. a 4. predpoklad).

3.5 Návrh modelu

Pri návrhu modelu bolo nutné zvoliť vhodný prístupu vyrovnávania zaťaženia a akým spôsobom odolávať chybám (v tomto prípade len chyby zrútením).

3.5.1 Výber prístupu vyrovnávania zaťaženia

Z hľadiska času, kedy získame atribúty popísané v sekcii 2.1.2, sú na výber tri prístupy: *statický*, *polostatický* a *dynamický*. Keďže jedna z vecí, ktorá nie je v predpokladoch, je znalosť časových zložitostí podúloh pred výpočtom a ich hodnoty sa dozvedáme až po spracovaní podúloh, použitý musí byť niektorý z *dynamických prístupov*.

Z hľadiska potreby centrálnej autority na riadenie procesu vyrovnávania zaťaženia je na výber prístup *centralizovaný* a *distribuovaný*. Procesory sú však všetky nespoľahlivé, a ani nie je predpoklad, že je aspoň jeden aktívny procesor v každom momente, takže výber sa zúžil na *distribuovaný* prístup. Tu ešte ostáva vybrať iniciátora vyrovnávania zaťaženia a spôsob výberu zdroja/cieľa presunu. Pri výbere iniciátora logicky vyplýva použitie typu iniciovaného prijímateľom, pretože v prostredí bude veľmi málo nevyťažených procesorov. Pri výbere zdroja/cieľa presunu sa zvolil výber všetkých procesorov (čiže procesor neadresuje požiadavku o podúlohy jednému procesoru, ale posiela požiadavku všetkým (broadcast)), pretože pri nespoľahlivom prostredí je vhodné informácie rozposielať všetkým, aby sa v čo najväčšej miere znížila redundancia(duplicita) výpočtu podúloh.

3.5.2 Výber spôsobu odolnosti voči chybe

Pri výbere spôsobu odolávania voči chybám bol kladený dôraz na zotavenie sa z poruchy, čo je v navrhovanom modeli najpodstatnejšia časť. Ostatné kroky procedúry odolnosti voči chybám sú nad rámec funkčnosti tohto modelu, preto ich prípadnú nevyhnutnosť je simulovaná.

Na riešenie zotavenia sa po výskyte chyby existujú dve metódy, *doprednéé zotavenie* a *spätné zotavenie*. Pri doprednom zotavení je snaha sa dostať z neplatného stavu do platného za behu, bez nutnosti návratu do predošlých platných stavov. Navrhovaný model sa má ale zotaviť zo zlyhania zrútením, čo znamená, že sa jeho beh zastaví a stratia sa tým aj všetky informácie uložené v dočasnej pamäti. Pri spätnom zotavení sa ale ukladá konzistentný stav výpočtu na disk a po výskyte chyby je možnosť sa následne z tohto bodu obnoviť, čo je pre navrhovaný model vhodné riešenie.

3.5.3 Popis fungovania modelu

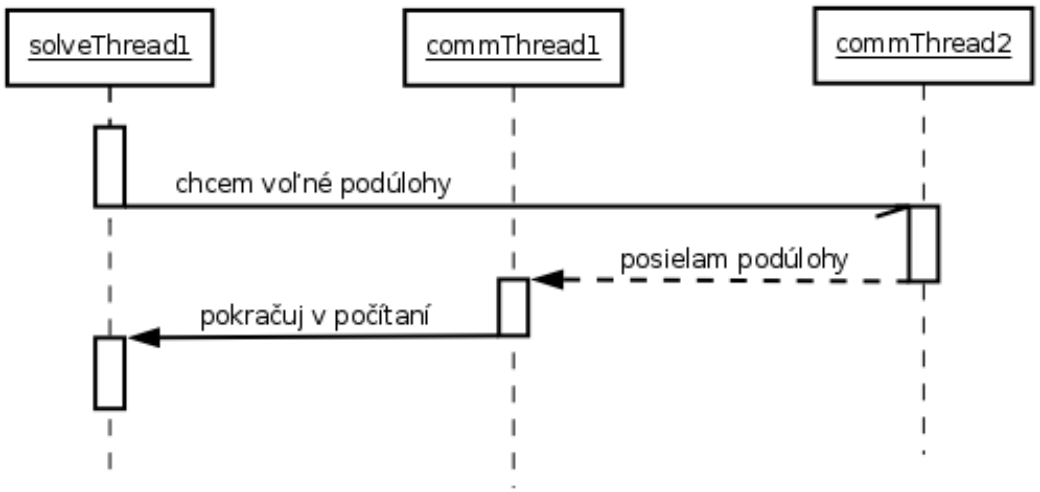
V nasledujúcej časti sa bude pracovať s nasledujúcimi pojmami:

- solveThread1 - výpočtové vlákno procesora 1, ktoré rieši samotné podúlohy.
- commThread1 - komunikačné vlákno procesora 1, ktoré sa stará o prijímanie správ a posielanie príslušných odpovedí na ne.
- commThread2 - komunikačné vlákno ľubovoľného procesora v sieti (iného ako je procesor 1).

Podúlohy majú tri možné stavy, buď sú už vyriešené, alebo sú nepridelené (voľné), alebo sú pridelené. Na začiatku výpočtu má u seba všetky podúlohy každý procesor, ale pridelené ich má len prvý z nich (teda prvý procesor je ich vlastníkom). Procesom vyrovnávania zaťaženia sa podúlohy postupne rozdelia medzi ostatné procesory. Vždy keď procesor pošle svoje pridelené podúlohy inému procesoru, tak po odoslaní prestáva byť ich vlastníkom a novým vlastníkom sa stáva procesor, ktorému sú adresované v momente, keď ich príjme. Takýmto prideľovaním podúloh procesorom je snaha o zabezpečenie, aby každá podúloha mala vlastníka, teda aby sa predišlo zbytočnej duplicite výpočtov. V prípade, že procesor zlyhá, stratí vlastnícke práva na všetky podúlohy, ktoré vlastnil, a stávajú sa nepridelenými (žiadny procesor ich nevlastní). Jediné, čo nestratí, sú podúlohy, ktoré už vyriešil. Pri tomto procese strácania vlastníctva, ale vznikajú podúlohy, ktoré sa stávajú nepridelenými. Tie sa následne prideľujú, až keď nie sú v sieti žiadne nevyriešené pridelené podúlohy, ktoré nemá pridelený žiadny procesor. Pri takomto modeli nastávajú viaceré situácie, ktoré sú v nasledujúcej časti popísané:

Procesor nemá voľnú podúlohu, ale v sieti sú

Keď solveThread dopočíta poslednú podúlohu, ktorú má procesor pridelenú, pošle do siete broadcast so žiadosťou o voľné podúlohy a pozastaví svoju činnosť. Keďže voľné podúlohy sa v sieti nachádzajú, určite niektoré prídu komunikačnému vláknu v správe, ktorá bola odpoveďou na žiadosť o voľné



Obr. 3.1: Procesor chce voľné podúlohy zo siete

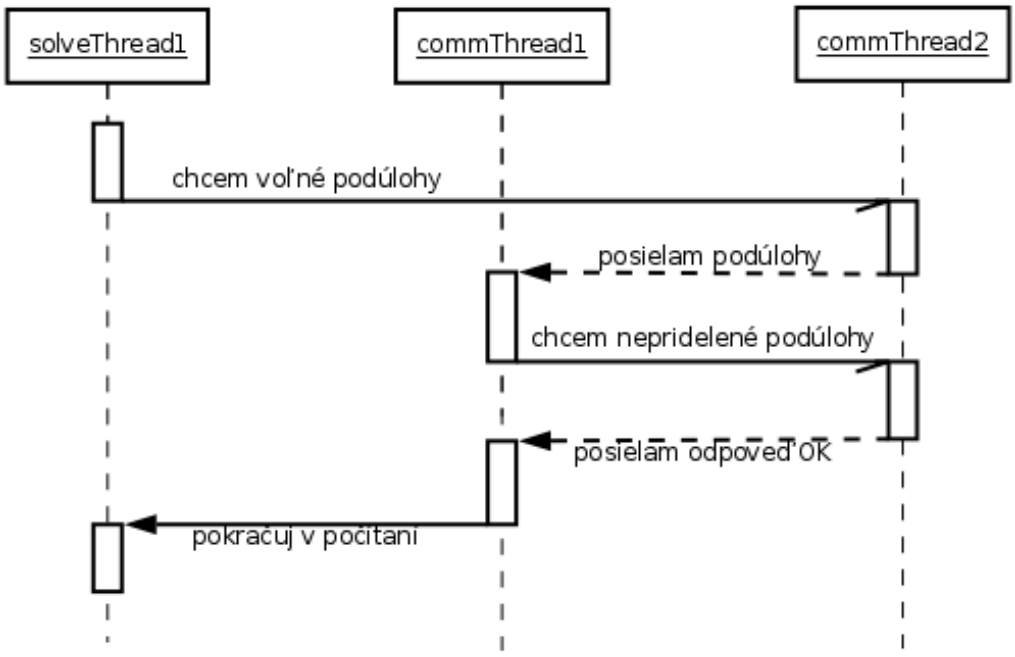
podúlohy. Komunikačné vlákno ich priradí do fronty a signalizuje vláknu solveThread, že výpočet môže pokračovať (obrázok 3.1).

Procesor nemá voľnú podúlohu, v sieti tiež nie je, ale sú nepridelené podúlohy

Postup začína ako v minulom prípade, len s tým rozdielom, že procesoru žiadny procesor neposlal podúlohy. Preto sa pozrie, či sú v sieti nepridelené podúlohy, a ak sú, tak posiela správu so žiadosťou, že si ich chce prideliť. Keď mu každý procesor odpovie, že si ich môže prideliť, pridelí si všetky nepridelené podúlohy do svojej fronty (obrázok 3.2).

Viac procesorov žiada o nepridelené podúlohy

Postup začína ako v minulom prípade, len s tým rozdielom, že o nepridelené podúlohy žiadajú viaceré procesory. Každý procesor má unikátne číslo (rank) v rámci siete, v ktorej sa nachádza. Čiže pri tejto situácii víťazí procesor s najväčším unikátnym číslom a on si pridelí všetky nepridelené podúlohy. Procesory, ktoré mali menší rank, následne posielajú broadcast so žiadosťou o voľné podúlohy (obrázok 3.3).



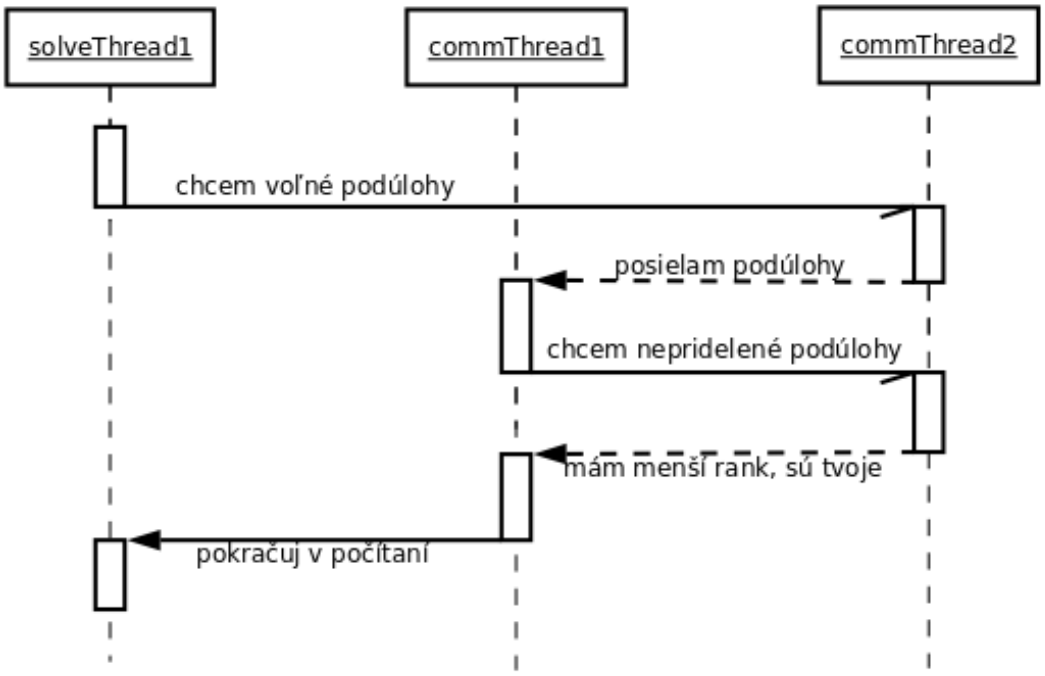
Obr. 3.2: Procesor chce nepridelené podúlohy

Procesor nezískal nepridelené podúlohy

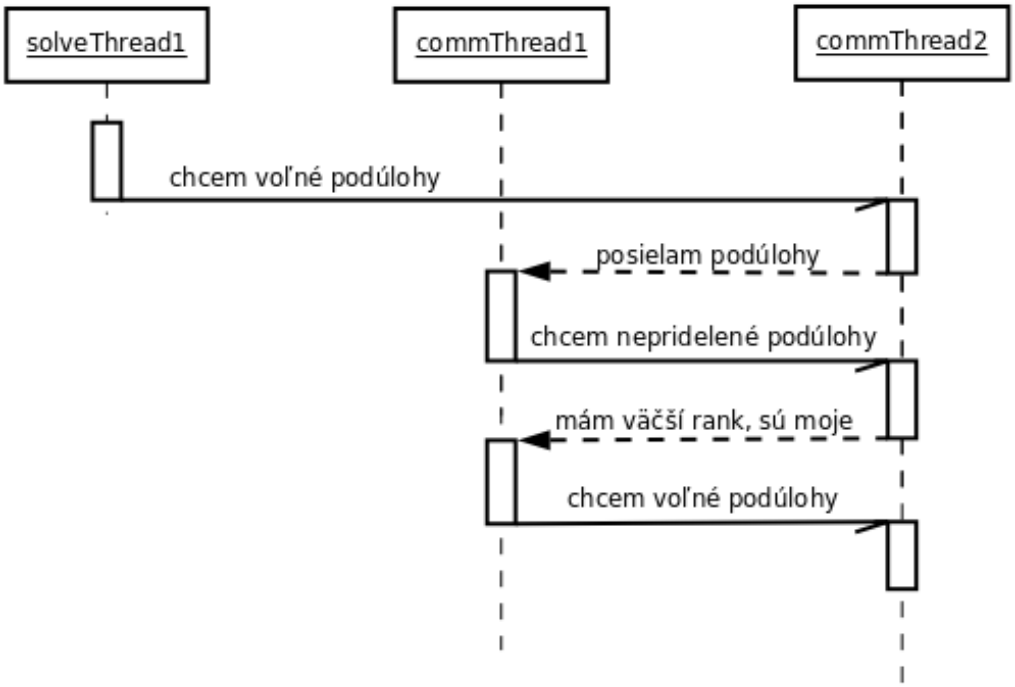
Postup začína ako v minulom prípade, len s tým rozdielom, že „boj" o nepridelené podúlohy procesor prehral. Keďže od procesora, ktorý získal nepridelené podúlohy, už je možné získať voľné podúlohy, procesor, ktorý nezískal nepridelené podúlohy, posiela broadcast so žiadosťou o voľné podúlohy (obrázok 3.4).

Procesor nemá voľnú podúlohu, v sieti nie je, a nie je ani nepridelená podúloha

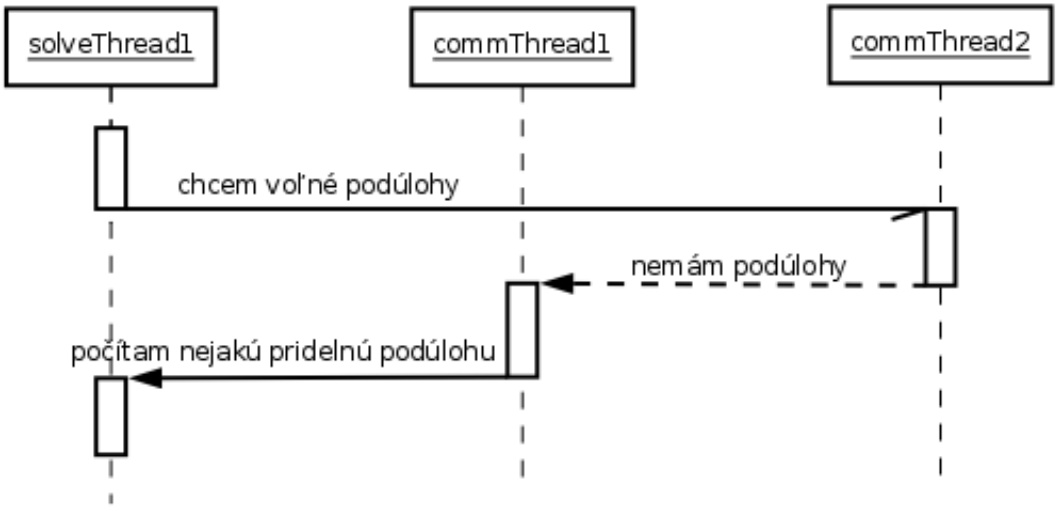
Keď solveThread dopočíta poslednú podúlohu, ktorú má procesor pridelenú, pošle do všetkým procesorom správu so žiadosťou o voľné podúlohy a pozastaví svoju činnosť. Ak v sieti už nie sú voľné podúlohy, procesoru od nikoho žiadne neprídu, tak sa pozrie, či sú nepridelené podúlohy, a keďže nie sú, začne riešiť náhodne vybratú priradenú podúlohu, ktorú aktuálne rieši iný procesor (obrázok 3.5).



Obr. 3.3: Viac procesorov žiada o nepridelené podúlohy



Obr. 3.4: Procesor nezískal nepridelené podúlohy

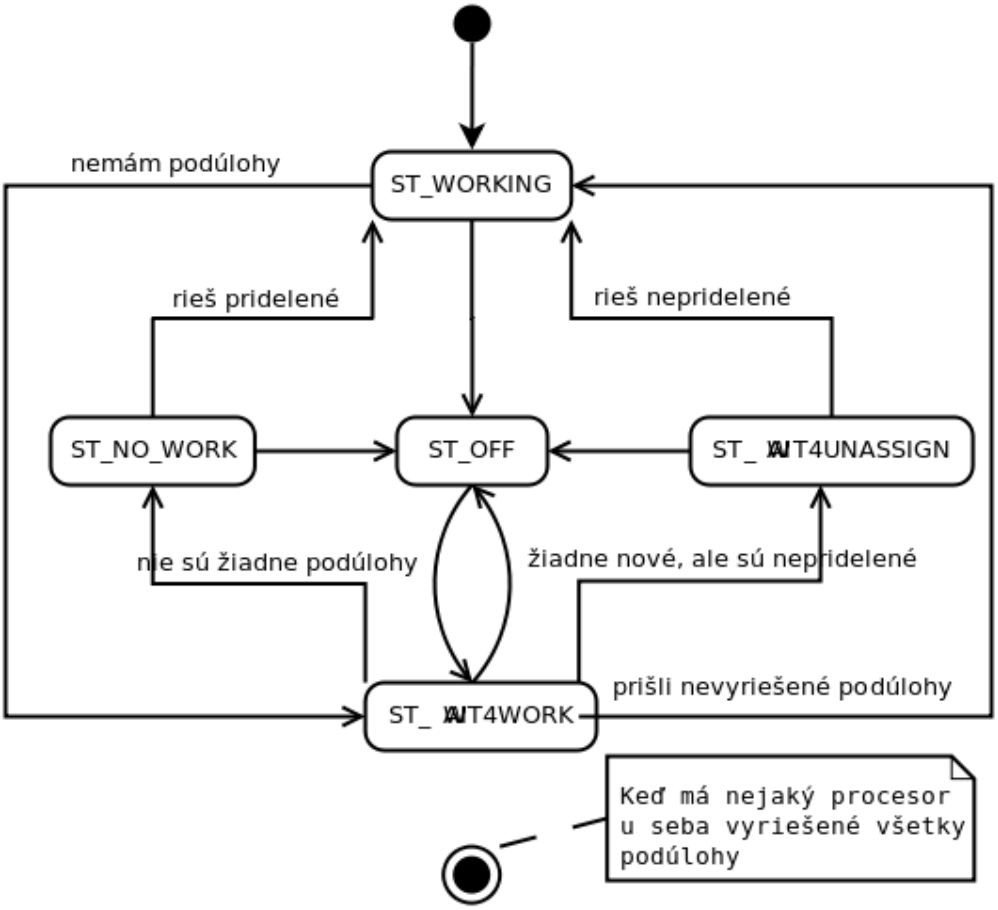


Obr. 3.5: Žiadne nepridelené podúlohy nie sú

Procesor zlyhal

V tomto prípade procesor stráca voľné podúlohy, teda si vymaže frontu pridelených podúloh a v tomto stave odpovedá na všetky prichádzajúce správy, že zlyhal. Po obnovení posiela všetkým procesorom v sieti správu so žiadosťou o voľné podúlohy.

Bližší popis, v akých stavoch sa môže model nachádzať a ako cez ne prechádza, ilustruje stavový diagram (obrázok 3.6).

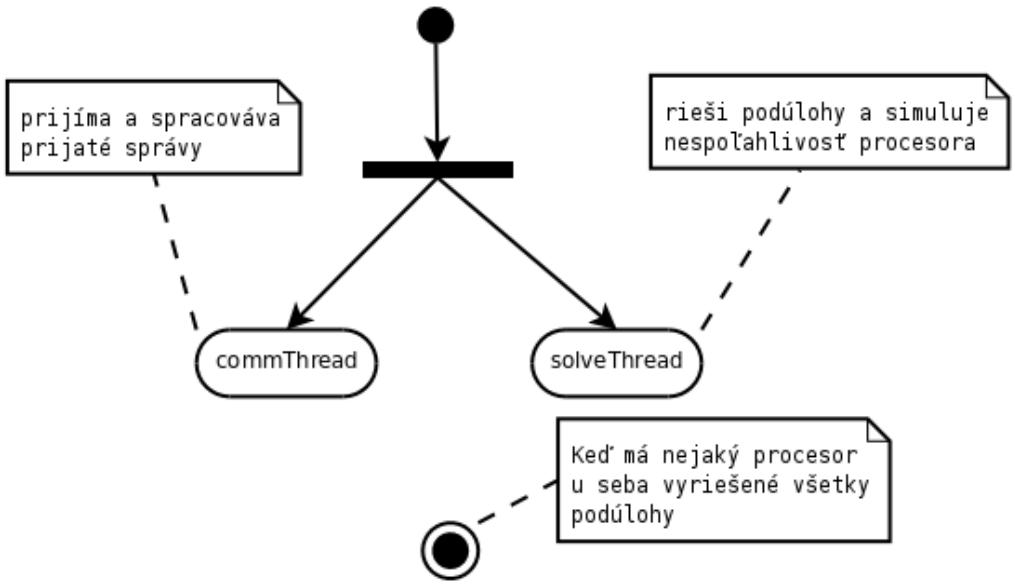


Obr. 3.6: UML stavový diagram

Kapitola 4

Implementácia modelu

Ako komunikačná knižnica bola použitá TPL(Thread Parallel Library)[Pla03], ktorá umožňuje vytváranie viac-vláknových aplikacií a je bezpečná voči vláknam (thread-safe). Je to knižnica pre programovací jazyk C, takže v tomto jazyku bol aj model implementovaný. Samotné vyrovnávanie zaťaženia je riešené ako dvoj-vláknová aplikácia, pričom jedno len prijíma správy a odpovedá na ne, a druhé rieši podúlohy (obrázok 4.1).



Obr. 4.1: UML diagram aktivít

4.1 Popis zdrojového kódu

Nasledujúce popisy riešenia implementácie modelu nie sú úplne presné kópie zdrojového kódu. Sú vyňaté len nosné časti, ktorých riešenie je z implementačného hľadiska zaujímavé.

4.1.1 Popis implementácie vyrovnávanie zaťaženia

V tejto sekcii je popísaná implementácia algoritmu vyrovnávania zaťaženia bez výpadkov.

```
solveThread(void)
 1: while (procStatus < ST_WAIT4END) {
 2:    if (takeFromTaskToDo(&actualTask) == EXIT_SUCCESS) {
 3:       solveTask(actualTask.value);
 4:    } else {
 5:      waitForCondition(INT_MAX,(myRank<input_cfg->nHwCores));
 6:    }
```

Vlákno *solveThread*[1] sa stará o výpočet podúloh. V cykle kontroluje, či nie je výpočet ukončený (1) (to je reprezentované stavmi *ST_WAIT4END* a *ST_END*). Ak má zoznam pridelených podúloh neprázdny, jednu podúlohu zo zoznamu vyberie, priradí ju do premennej *actualTask* (2) a vyrieši ju (3). Ak je ale prázdny, musí čakať (maximálne *INT_MAX*=2147483647 sekúnd) (5), kým mu druhé vlákno neoznámi, že môže pokračovať (buď procesor získal voľné podúlohy, alebo prišla správa o ukončení).

```
solveTask(int taskTime)
 1: if(waitForCondition(taskTime,myRank < input_cfg->nHwCores)==true){
 2:    if((getToDoTaskCount() == 0) && (procStatus < ST_WAIT4END)){
 3:        changeStatus(ST_WAIT4WORK);
 4:    }
 5:    return;
 6: }
 7: actualTask.status = TASK_SOLVED;
 8: addToSolvedTasks(&actualTask);
 9: if (areAllSolved()) {
```

[1] V nasledujúcej časti je vlákno *solveThread* označované výpočtové vlákno

```
10:    changeStatus(ST_WAIT4END);
11: } else
12:    if (getToDoTaskCount() == 0) {
13:       changeStatus(ST_WAIT4WORK);
14:    }
```

Funkcia *solveTask* má na starosti samotný výpočet podúlohy. Simulácia výpočtu sa udeje vo funkcii *waitForCondition* (1), ktorá je popísaná nižšie. Výpočet v tejto funkcii môže byť prerušený (návratová hodnota = *true*) a to buď z dôvodu, že iný procesor v sieti mu poslal výsledok podúlohy, ktorú práve rieši (ak to bola posledná pridelená podúloha, posiela všetkým procesorom správu so žiadosťou o nevyriešené podúlohy (3)), alebo prišla správa o ukončení výpočtu a funkcia tým pádom končí (5). V prípade, že nenastala ani jedna z týchto situácií, znamená to, že funkcia podúlohu úspešne vyriešila. Podúlohu si označí ako vyriešenú (7). Ak to bola posledná vyriešená podúloha celej úlohy, zmení stav na *ST_WAIT4END* a posiela všetkým procesorom správu *MSG_END* o ukončení výpočtu (10), ak nie, skontroluje si zoznam svojich podúloh, a ak v ňom žiadna nie je, zmení stav na *ST_WAIT4WORK* a posiela všetkým procesorom správu *MSG_NEED_WORK* so žiadosťou o nevyriešené podúlohy (13).

```
waitForCondition(wait_time, activeWorkSimulation)
 1: waitCondition = false;
 2: while ((wait_time > 0) && (waitCondition == false)) {
 3:    if (!activeWorkSimulation) {
 4:       if (wait_time < FALLWAITSTEP) {
 5:          usleep(wait_time*1000000);
 6:       } else {
 7:          usleep(FALLWAITSTEP*1000000);
 8:       }
 9:    } else {
10:       if (wait_time < FALLWAITSTEP) {
11:           empty_work(wait_time);
12:       } else {
13:           empty_work(FALLWAITSTEP);
14:       }
15:    }
16:    wait_time -= FALLWAITSTEP;
17: }
```

```
19: return(waitCondition);
```

Funkcia *waitForCondition* čaká zadaný čas, kým nie je splnená podmienka *waitCondition* (2). Následne podľa premennej *activeWorkSimulation*, buď čaká „aktívne", teda skutočne vyťažuje procesor pomocou funkcie *empty_work*[1], alebo čaká „pasívne", bez vyťaženia procesora, čo vykonáva funkciou *usleep*. Ktorá sa použije závisí od hodnoty parametra *NR_CORES* vo vstupnom súbore. Kontrola premennej *waitCondition* sa vykonáva v časových intervaloch daných konštantou *FALLWAITSTEP*.

Podmienkou *waitCondition* oznamuje komunikačné vlákno výpočtovému, že nastala zmena, ktorá okamžite ovplyvňuje ďalší beh výpočtového vlákna. Na hodnotu true podmienku nastavuje komunikačné vlákno v nasledovných troch prípadoch:

1. *Komunikačné vlákno zmenilo stav na ST_WORKING.* - to znamená, že výpočtové vlákno čakalo na nové nevyriešené podúlohy a keď mu prišli, môže pokračovať vo výpočte.

2. *Komunikačné vlákno zmenilo stav na ST_WAIT4END.* - iný procesor poslal správu, že už má vyriešenú celú úlohu, čiže výpočtové vlákno preruší na aktuálny výpočet a skončí.

3. *Iný procesor poslal výsledok podúlohy, ktorú výpočtové vlákno práve počíta.* - výpočtové vlákno už nemusí ďalej pokračovať vo výpočte podúlohy, ktorej výsledok mu prišiel a môže pokračovať na ďalšej podúlohe.

Funkcia má návratovú hodnotu *true*, ak jej beh bol prerušený podmienkou *waitCondition*, inak *false*.

```
sendMessage(recepient, MSG_NEED_WORK)
 1: amount = getRecentlySolvedTasksCount();
 2: tpl_pkint(message, &amount, 1);
 3: packRecentlySolvedTasks(message, amount);
 4: tpl_send(procIDArray, input_cfg->nProcs-1, tag, message);
```

[1]Funkcia *empty_work(N)* vyťažuje procesor N sekúnd, pričom ten čas nie je rozdiel medzi zavolaním funkcie a jej skočením, ale procesorový čas, teda koľko sekúnd sa musí reálne procesor venovať tomu procesu, v ktorom táto funkcia beží. Návratová hodnota funkcie obsahuje časový rozdiel medzi zavolaním funkcie a jej skočením.

Pri posielaní správy *MSG_NEED_WORK* všetkým procesorom so žiadosťou o voľné podúlohy (4), funkcia k správe pribalí aj podúlohy, ktoré výpočtové vlákno vyriešilo od chvíle, keď žiadalo o podúlohy naposledy (3). Výpočtové vlákno totiž neposiela výsledok podúlohy hneď ako ju vypočíta, aby nadmerne nezaťažoval sieť posielaním správ.

```
communicationThread(void)
 1: while (procStatus != ST_END) {
 2:    tpl_recv(matchAll, &send_from, &tag, &message);
 3:    simulateLatency(message, send_from);
 4:    if (tag == MSG_END) {
 5:        handleMSG_END(send_from);
 6:    } else {
 7:       if (procStatus < ST_WAIT4END) {
 8:          switch( tag ) {
 9:             case MSG_ON:
10:                handleMSG_ON(send_from, message);break;
11:             case MSG_NEED_WORK:
12:                handleMSG_NEED_WORK(send_from, message);break;
13:             case MSG_TASKS:
14:                handleMSG_TASKS(send_from, message, false);break;
15:             case MSG_ALLTASKS:
16:                handleMSG_TASKS(send_from, message, false);break;
17:             case MSG_WANT_UNASSIGN:
18:                handleMSG_WANT_UNASSIGN(send_from, message);break;
19:             case MSG_ANSWER:
20:                handleMSG_ANSWER(send_from, message, false);break;
21:             default:
22:                 exit();
23:          }
24:       }
25:    }
26:    tpl_destroy(message);
27: }
```

Vlákno *communicationThread*[1] má na starosti príjem a spracovanie správ. Komunikačné vlákno čaká pomocou funkcie *tpl_recv* na správu (3). Táto funkcia

[1]V nasledujúcej časti je vlákno *communicationThread* označované komunikačné vlákno

je blokujúca, takže vo vykonávaní ďalších príkazov bude pokračovať, až keď tomuto procesoru bude adresovaná ľubovoľná správa. Po prijatí správy určenej pre procesor, ak je zadané nenulové oneskorenie, simuluje oneskorenie správy a pokračuje spracovaním správy podľa označenia (*tag*) správy a aktuálneho stavu procesora. Stav procesora totiž rozhoduje, či sa vôbec správa spracuje, alebo nie. Ak sa procesor nachádza v stave reprezentujúcom ukončenie výpočtu úlohy (*ST_WAIT4END* alebo *ST_END*), na správy ďalej neodpovedá, len čaká, aby procesor mohol korektne ukončiť beh. Komunikačné vlákno je aktívne až do momentu nastavenia stavu procesora *ST_END*.

```
handleMSG_NEED_WORK(int send_from, void *message)
 1: unpackMessage(message);
 2: if (areAllSolved()) {
 3:    changeStatus(ST_WAIT4END);
 4: } else {
 5:    sendMessage(send_from, MSG_TASKS);
 6: }
```

Po prijatí žiadosti o voľné podúlohy funkcia najprv správu rozbalí (1). Následne skontroluje, či už nemá všetky podúlohy vyriešené a ak áno, zmení stav na *ST_WAIT4END* a pošle všetkým procesorom správu *MSG_END* o ukončení výpočtu (3). Ak úloha nie je vyriešená, pošle funkcia procesoru, ktorý žiadal voľné podúlohy, správu *MSG_TASKS* (5)

```
sendMessage(recepient, MSG_TASKS)
 1: if (getToDoTaskCount() > 0) {
 2:    amount = getAmountOfTasks();
 3:    tpl_pkint(message, &amount, 1);
 4:    packTasksToDo(message, amount);
 5: } else {
 6:    amount = getSolvedTasksCount();
 7:    if (procStatus == ST_WORKING) {
 8:       amount2 = amount + 1;
 9:       tpl_pkint(message, &amount2, 1);
10:       packActualTask(message);
11:    } else {
12:       tpl_pkint(message, &amount, 1);
13:    }
```

```
14:    packAllSolvedTasks(message, amount);
15: }
16: tpl_send(&recepient, 1, tag, message);
```

Pri posielaní správy *MSG_TASKS* sa pribaľujú k správe nevyriešené podúlohy, ktoré má pridelené. Ak má zoznam pridelených podúloh neprázdny (1), tak istý počet z nich (ich počet je riadený vstupným parametrom -a)(2) zabalí do správy (4) a odošle ju procesoru, ktorý žiadal o podúlohy (16). Ak v zozname žiadne nemá, pribalí k správe všetky vyriešené podúlohy, ktoré má (14) a v prípade, že je podúloha, na ktorej práve pracuje, tak pribalí aj tú (10), ale označí jej stav ako *TASK_ASSIGN*. To znamená, že táto podúloha nie je ani vyriešená, ani voľná, ale iný procesor túto úlohu práve vykonáva.

```
handleMSG_TASKS(send_from, *message)
 1: unpackMessage(message);
 2: if (areAllSolved()) {
 3:    changeStatus(ST_WAIT4END);
 4: }
 5: if (getToDoTaskCount() > 0) {
 6:    if (procStatus != ST_WORKING) {
 7:       changeStatus(ST_WORKING);
 8:    }
 9: } else {
10:    answerArray[send_from] = ANS_YES;
11:    if (areAllYES()) {
12:       if (getFirstAssignTask() == EXIT_SUCCESS) {
13:          changeStatus(ST_WORKING);
14:       }
15:    }
16: }
```

Ak vyriešené podúlohy, ktoré má procesor, spolu s vyriešenými, ktoré prišli v správe *MSG_TASKS*, tvoria riešenie celej úlohy, tak procesor zmení stav na *ST_WAIT4END* a posiela správu *MSG_END* všetkým procesorom (3). V prípade, že úloha nie je vyriešená a z podúloh, ktoré prišli je aspoň jedna nevyriešená, funkcia zmenou stavu na *ST_WORKING* signalizuje výpočtovému vláknu, že môže pokračovať vo výpočte (7). Ak ale procesoru neprišli žiadne nevyriešené podúlohy, poznačí si len, že odosielateľ už svoju odpoveď poslal (10). Keď príde

správa *MSG_TASKS* od každého procesora v sieti a žiadny z nich neposlal nevyriešenú podúlohu, pridelí si prvú podúlohu, na ktorej práve pracuje iný procesor (8) (teda takú, ktorá je v stave *TASK_ASSIGN*) a zmení stav na *ST_WORKING* (13).

```
handleMSG_END(send_from)
 1: if (procStatus != ST_WAIT4END) {
 2:    changeStatus(ST_WAIT4END);
 3: }
 4: answerArray[send_from] = ANS_YES;
 5: if (areAllYES()) {
 6:    changeStatus(ST_END);
 7: }
```

Keď príde procesoru správa o ukončení výpočtu *MSG_END*, označí si odosielateľa správy, že už poslal správu o ukončení výpočtu (4). Ak procesor nebol po prijatí správy ešte v stave *ST_WAIT4END*, zmení ho na tento stav a posiela všetkým procesorom správu *MSG_END*. To je zároveň posledná správa, ktorú procesor pošle. Po spracovaní správ *MSG_END* od všetkých procesorov prejde do stavu *ST_END* (6) a ukončí svoj beh.

Použitá komunikačná knižnica vyžaduje, aby procesoru, ktorý ukončil svoj beh, už nebola doručená žiadna správa. Teda bolo potrebné vyriešiť korektné ukončovanie programu. Keďže ale posielanie správ je asynchrónne, nie je zaručené, že správa posielaná v čase T_1 procesorom 1 procesoru 3 príde skôr, ako správa posielaná v čase T_2 procesorom 2 procesoru 3, pričom $T_1 < T_2$. Z čoho vyplýva, že procesor nemôže skončiť hneď po zistení výsledku úlohy. O čo sa dá ale oprieť, je poradie správ pri komunikácii medzi dvojicou procesorov. Teda správa posielaná v čase T_1 procesorom 1 procesoru 2 príde skôr ako správa posielaná v čase T_2 procesorom 1 procesoru 2, pričom $T_1 < T_2$. Riešenie problému je teda nasledovné: Keď sa procesor dozvie, že úloha je vyriešená, posiela všetkým procesorom správu o ukončení výpočtu. Táto správa je ale poslednou, ktorú procesor počas svojho behu pošle. A keď procesoru prídu správy o ukončení výpočtu od všetkých ostatných procesorov, je zaručené, že žiadna ďalšia správa mu nie je adresovaná a môže svoj beh korektne ukončiť.

4.1.2 Popis implementácie nespoľahlivosti

V tejto časti sú popísané časti kódu, ktorých pridaním simulujeme nespoľahlivosť procesora a zároveň sú riešené výpadky s tým spojené.

```
solveThread(void)
 1: while (procStatus < ST_WAIT4END) {
 2:    if (nextFall < 0) {
 3:       changeStatus(ST_OFF);
 4:       nextFall = INT_MAX;
 5:       waitForCondition(input_cfg->fallDur[myRank][current_fall],
   (myRank <input_cfg->nHwCores));
 6:       if (procStatus < ST_WAIT4END) {
 7:          changeStatus(ST_ON);
 8:       }
 9:       current_fall = ((current_fall+1)%RANDOMFALLSCOUNT);
10:       nextFall = input_cfg->fallProbs[myRank][current_fall];
11:    }
12:    if (takeFromTaskToDo(&actualTask) == EXIT_SUCCESS) {
13:       solveTask();
14:    } else {
15:       waitForCondition(INT_MAX, (myRank < input_cfg->nHwCores));
16:    }
17: }
```

Pri nespoľahlivých procesoroch nám vo výpočtovom vlákne pribudla premenná *nextFall*, ktorá obsahuje čas do najbližšieho zlyhania.

Ak ešte zostáva čas do zlyhania (2), tak výpočtové vlákno pokračuje v riešení podúloh (13), alebo ak nemá žiadne ďalšie pridelené podúlohy, čaká (15) na nové, prípadne, či nenastane zlyhanie. V prípade, že už čas do zlyhania vypršal (2), výpočtové vlákno zmení stav na *ST_OFF* (3) a simuluje zlyhanie (5). To bude trvať *input_cfg->fallDur[myRank][current_fall]* sekúnd. Po uplynutí času zlyhania, zmení stav na *ST_ON* a posiela všetkým procesorom správu *MSG_ON*, že sa zobudil (7). Do premennej *nextFall* uloží čas do ďalšieho zlyhania (10).

```
solveTask(int taskTime)
  ...
  if (nextFall < 0) { return; }
  ...
```

Funkcia *solveTask* bude fungovať rovnako ako pri spoľahlivých procesoroch, len pribudne medzi riadky 1 a 2 jedna podmienka, ktorá súvisí so simuláciou zlyhania a preruší beh funkcie.

```
waitForCondition(wait_time, activeWorkSimulation)
 1: waitCondition = false;
 2: while ((wait_time > 0) && (waitCondition == false)) {
 3:    if (!activeWorkSimulation) {
 4:       if (wait_time < FALLWAITSTEP) {
 5:          usleep(wait_time*1000000);
 6:          nextFall -= wait_time;
 7:       } else {
 8:          usleep(FALLWAITSTEP*1000000);
 9:          nextFall -= FALLWAITSTEP;
10:       }
11:    } else {
12:       if (wait_time < FALLWAITSTEP) {
13:          nextFall -= empty_work(wait_time);
14:       } else {
15:          nextFall -= empty_work(FALLWAITSTEP);
16:       }
17:    }
18:    if (nextFall < 0) {
19:        waitCondition = true;
20:    }
21:    wait_time -= FALLWAITSTEP;
22: }
23: return(waitCondition);
```

Tiež funkcia *waitForCondition* sa mierne zmení. Pri každom cykle sa od premennej naxtFall odpočíta čas, ktorý trvala funkcia na simuláciu práce (*usleep*, alebo *empty_work*). Ak je nextFall už menšie ako 0 (18), znamená to, že čas do ďalšieho zlyhania uplynul a podmienka *waitCondition* sa nastaví na true (19), čím sa výpočet preruší.

```
sendMessage(recepient, MSG_ON)
 1: amount = getSolvedTasksCount();
 2: tpl_pkint(message, &amount, 1);
 3: packAllSolvedTasks(message, amount);
 4: tpl_send(procIDArray, input_cfg->nProcs-1, tag, message);
```

Do správy *MSG_ON* sa pribalia všetky vyriešené podúlohy (3), čo má procesor u seba a pošle správu všetkým procesorom.

```
handleMSG_ON(int send_from, void *message)
 1: unpackMessage(message);
 2: if (areAllSolved()) {
 3:    changeStatus(ST_WAIT4END);
 4: } else {
 5:    sendMessage(send_from, MSG_ALLTASKS);
 6: }
```

Ak po prijatí správy *MSG_ON* funkcia zistí, že už sú všetky podúlohy vyriešené (2), zmení stav na *ST_WAIT4END* a pošle všetkým procesorom v sieti správu *MSG_END* (3), v opačnom prípade posiela správu *MSG_ALLTASKS* späť odosielateľovi (5).

```
sendMessage(recepient, MSG_ALLTASKS)
 1: if (getToDoTaskCount() == 0) {
 2:    amount = 0;
 3: } else {
 4:    amount = getAmountOfTasks();
 5: }
 6: amount2 = amount + getSolvedTasksCount();
 7: if (procStatus == ST_WORKING) {
 8:    amount2 = amount2 + 1;
 9:    tpl_pkint(message, &amount2, 1);
10:    packActualTask(message);
11: } else {
12:    tpl_pkint(message, &amount2, 1);
13: }
14: packTasksToDo(message, amount);
15: packAllSolvedTasks(message, getSolvedTasksCount());
16: tpl_send(&recepient, 1, tag, message);
```

Do správy *MSG_ALLTASKS* pribalí podúlohy, ktoré má pridelené (14). Ich počet určí podobne ako pri posielaní správy *MSG_TASKS* (4). Ďalej pribalí všetky vyriešené podúlohy (15), aby mal odosielateľ, čo najaktuálnejší prehľad, ktoré podúlohy sú vyriešené a ktoré nie. Ak procesor práve pracuje na podúlohe, tak pribalí aj tú (10).

```
communicationThread(void)
...
if (procStatus == ST_OFF || tag == MSG_OFF) {
    handleMSG_OFF(tag, send_from, message);
} else {
...
```

V prípade komunikačného vlákna sa vloží medzi riadky 6 a 7 „spoľahlivej" verzie funkcie *communicationThread* vyššie uvedená podmienka. Táto podmienka simuluje správanie zlyhaného procesora v sieti. Spracováva všetky správy, keď je procesor zlyhaný, teda v stave *ST_OFF* a tiež spracuje správu *MSG_OFF* od iného procesora.

```
handleMSG_OFF(int tag, int send_from, void *message)
 1: if (procStatus == ST_OFF) {
 2:    if (tag==MSG_ON||tag==MSG_NEED_WORK||tag==MSG_WANT_UNASSIGN) {
 3:      sendMessage(send_from, MSG_OFF);
 4:    }
 5: } else {
 6:    if (procStatus == ST_WAIT4WORK) {
 7:       handleMSG_TASKS(send_from, message, true);
 8:    } else {
 9:       if (procStatus == ST_WAIT4UNASSIGN) {
10:          handleMSG_ANSWER(send_from, message, true);
11:       }
12:    }
13: }
```

Keď procesor zlyhá, neposiela nikomu žiadnu správu o tom, že zlyhal. Ostatné procesory sa to dozvedia až vtedy, keď očakávajú od zlyhaného procesora odpoveď na ich žiadosť. A teda keď zlyhaný procesor dostane takú správu od iného procesora, na ktorú procesor očakáva odpoveď, pošle mu správu *MSG_OFF* (3). V prípade, že procesor obdrží správu od zlyhaného procesora *MSG_OFF*, v závislosti od toho v akom stave sa práve nachádza, zavolá príslušnú funkciu na spracovanie správy: (7), alebo (10). Posledný parameter pri volaní funkcie nastaví na true, čím oznamuje funkcii, že správa je od zlyhaného procesora.

```
handleMSG_TASKS(int send_from, void *message, _Bool isOff)
 1: if (isOff == false) {
 2:    unpackMessage(message);
 3:    if (areAllSolved()) {
 4:       changeStatus(ST_WAIT4END);
 5:    }
 6: }
 7: if (getToDoTaskCount() > 0) {
 8:    if (procStatus != ST_WORKING) {
 9:       changeStatus(ST_WORKING);
10:    }
11: } else {
12:    answerArray[send_from] = ANS_YES;
13:    if (areAllYES()) {
14:       if (isUnassignTask()) {
15:          if (input_cfg->takeAll == true) {
16:             handleMSG_ANSWER(send_from, message, true);
17:          } else {
18:             changeStatus(ST_WAIT4UNASSIGN);
19:          }
20:       } else {
21:          if (getFirstAssignTask() == EXIT_SUCCESS) {
22:             changeStatus(ST_WORKING);
23:          }
24:       }
25:    }
26: }
```

Zmena funkcie oproti pôvodnej je pridanie podmienky (1), pretože funkciu *handleMSG_TASKS* môže zavolať funkcia *handleMSG_OFF* s prázdnou správou. Spracovanie správy teda preskočíme a poznačíme si, že procesor poslal správu na našu žiadosť (12).

Ďalšia zmena sa týka skutočnosti, že procesor po zlyhaní stráca svoje pridelené podúlohy. Z čoho vyplýva, že nám vznikajú podúlohy, ktoré žiadny procesor nevlastní. Procesory sa musia preto dohodnúť, kto si ju pridelí. Takže, ak procesor nedostal žiadnu voľnú podúlohu, skontroluje, či existujú nepridelené podúlohy (14). Ak nie je, pridelí si jednu z podúloh, ktorá je v stave *TASK_ASSIGN* (21) a zmení stav na *ST_WORKING*. Naopak ak existujú nepri-

delené podúlohy, potom podľa parametra *input_cfg->takeAll* (15), buď si všetky nepridelené podúlohy pridelí bez akejkoľvek dohody (16), alebo zmení stav na *ST_WAIT4UNASSIGN* a posiela všetkým procesorom správu *MSG_WANT_UNASSIGN*, aby mu procesory odpovedali, či si ju môže prideliť alebo nie (18).

```
handleMSG_WANT_UNASSIGN(int send_from, void *message)
 1: if (getToDoTaskCount() == 0) {
 2:    sendMessage(send_from, MSG_ANSWER);
 3: } else {
 4:    sendMessage(send_from, MSG_TASKS);
 5: }
```

Ak procesor nemá žiadne pridelené podúlohy, posiela správu *MSG_ANSWER*, ale ak mu medzitým prišli nevyriešené podúlohy, pribalí ich k správe a posiela správu *MSG_TASKS*

```
sendMessage(recepient, MSG_ANSWER)
 1: if ((procStatus == ST_WAIT4UNASSIGN) && (myRank > recepient)) {
 2:    answer = ANS_NO;
 3: } else {
 4:    answer = ANS_YES;
 5: }
 6: tpl_pkint(message, &answer, 1);
 7: tpl_send(&recepient, 1, tag, message);
```

Ak procesor, ktorý je tiež v stave *ST_WAIT4UNASSIGN* a má unikátne číslo (rank) väčšie ako rank žiadateľa o nepridelenú prácu, posiela mu správu *MSG_ANSWER* s odpoveďou *ANS_NO*, inak *ANS_YES*

```
handleMSG_ANSWER(int send_from, void *message, _Bool isOff)
 1: if (isOff == false) {
 2:    tpl_upkint(message, &ans, 1);
 3: } else {
 4:    ans = ANS_YES;
 5: }
 6: answerArray[send_from] = ans;
 7: if (areAllYES()) {
 8:    if (getUnassignTasks() > 0) {
 9:       changeStatus(ST_WORKING);
```

```
10:     } else {
11:        if (getFirstAssignTask() == EXIT_SUCCESS) {
12:           changeStatus(ST_WORKING);
13:        }
14:     }
15: } else {
16:     if (areAllAnswered()) {
17:        changeStatus(ST_WAIT4WORK);
18:     }
19: }
```

Keďže môže byť aj táto funkcia zavolaná z funkcie *handleMSG_OFF*, podmienka (1) určí, či je v správe odpoveď na žiadosť, alebo správa prišla od zlyhaného procesora a teda odpoveď je automaticky *ANS_YES*. Ak už prišli správy od všetkých a nie sú všetky *ANS_YES* (7), tak v sieti musel byť aspoň jeden procesor, ktorý mal rank väčší ako on a získal nepridelené podúlohy. Zmení si svoj stav na *ST_WAIT4WORK* a posiela všetkým procesorom správu *MSG_NEED_WORK* (17). Ak ale od všetkých dostal odpoveď *ANS_YES*, tak ak sú ešte nepridelené podúlohy, tak si ich pridelí (8) a zmení stav na *ST_WORKING* (9), ak nie, tak si vezme jednu, na ktorej robí iný procesor (11) a zmení stav na *ST_WORKING* (12). Nepridelená práca môže zmiznúť, ak procesor, ktorý mal aktuálne nepridelené podúlohy pôvodne ich síce vyriešil, ale predtým ako stihol poslať ich výsledky zlyhal. Potom sa v priebehu dohadovania o nepridelené podúlohy zobudil a následne výsledky podúloh poslal všetkým, čiže z nepridelených podúloh sa stali vyriešené.

4.2 Vývojové prostredie a použité nástroje

Operačný systém: GNU/Linux

Distribúcia: Linux Mint 8

Jadro: 2.6.31

Programovací jazyk: C

Paralelizácia: Pthreads, TPL (Thread parallel library) [Pla03]

Kompilátor: gcc 4.4.1

Vývojové prostredie: Eclipse 3.5.1 + debugger GDB

Iné: Simulátor distribuovaného systému MPS [Pau07]

Kapitola 5

Experimenty

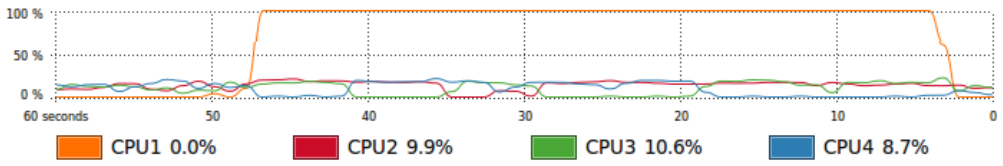
5.1 Testovacie prostredie

Testy prebiehali na nasledovnom PC:

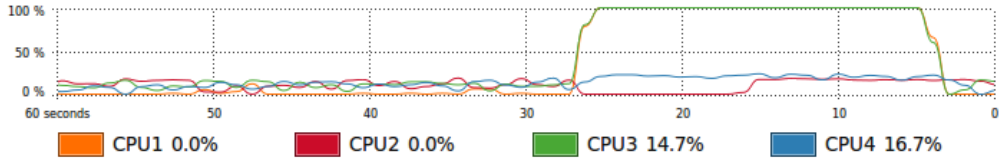
- Procesor: AMD Athlon X4 2.6 GHz (obsahuje 4 plnohodnotné jadrá)
- Operačný systém: Linux Mint 8
- Jadro OS: 2.6.31-14

Na simuláciu paralelného prostredia bol použitý simulátor MPS. Program bol spúšťaný s najväčšou prioritou 19, príkazom *nice*. Použitý operačný systém vie plne využit'potenciál použitého procesora pri použití viac-vláknových aplikacií, čo dokazujú aj screenshoty:

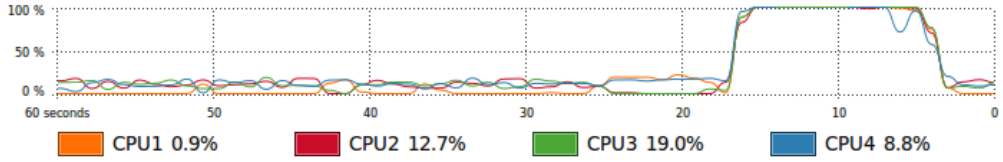
- Pri spustení programu s nastaveným jedným procesorom (parametre -p 1 -c 1)



- Pri spustení programu s nastavenými dvomi procesormi (parametre -p 2 -c 2)



- Pri spustení programu s nastavenými štyrmi procesormi (parametre -p 4 -c 4)



Čo je dôležité pripomenúť, je že navrhovaný model nerieši konkrétnu úlohu (teda nejaký výpočet), ale každá podúloha je len časový úsek, počas ktorého program „čaká" (simuluje časť konkrétneho výpočtu). A môže čakať „aktívne" (teda procesor sa reálne zaťažuje), alebo „spí" (procesor nie je zaťažený).

Za druhé je dôležité spomenúť, že výsledky, čo sa týka výsledných časov nezávisia len od frekvencie, závisia aj od použitej architektúry testovacieho systému (rôzne architektúry nepracujú s procesmi rovnako, napr. plánovanie procesov (scheduling)). Tiež je rozdiel či máme v počítači jeden procesor, viac procesorov, alebo máme distribuovanú sieť počítačov. Tento fakt významným spôsobom ovplyvňuje oneskorenie správ posielaných medzi procesormi. Z toho vyplýva, že nie je podstatné len použité testovacie prostredie a aký bude výsledný čas, pretože efektívnosť bude na rôznych architektúrach odlišná. Dôležité sú aj iné atribúty, ako počet poslaných správ medzi procesormi a pri našom modeli s nespoľahlivými procesormi aj počet duplicitných výpočtov. Pri navrhovanom modeli bola snaha o minimalizáciu duplicitných výpočtov a počtu správ posielaných medzi procesormi.

5.2 Parametre programu

Konštanty programu

Nachádzajú sa v súbore globals.h. Pri zmene konštanty je samozrejme nutné program prekompilovať.

Konštanta	**Popis konštanty**
RANDOMFALLSCOUNT	Počet náhodne vygenerovaných zlyhaní a ich trvaní. Pri viacerých zlyhaniach ako RANDOMFALLSCOUNT sa z posledného znovu začína na prvom.
MAXPROCCOUNT	Maximálny počet procesorov
MAXTASKCNT	Maximálny počet podúloh
MINFALLTIME	Minimálny časový úsek do ďalšieho zlyhania
MINDURTIME	Minimálne trvanie zlyhania
MINTASKCOMPLEXITY	Minimálna veľkosť podúlohy
FALLWAITSTEP	Čas medzi kontrolou podmienky, ktorá signalizuje, že vo výpočte sa môže pokračovať, alebo sa môže skončiť výpočet.

Vstupné parametre

Param.	Popis parametra	Predvolené
`lbmodel gen-input [OPTIONS]`		
-h	vypíše pomoc	-
-o	súbor, do ktorého sa vstupné parametre uložia *(povinný)*	-
-i	súbor, z ktorého sa doplnia parametre, ktoré užívateľ nezadal	-
-c	počet aktívnych procesorových jadier v PC	1
-d	oneskorenie posielaných správ	0.0
-a	počet pridelených podúloh posielaných medzi procesormi (one = 1, npart = n-tina z pridelených, all = všetky)	npart
-f	vynulovanie už vypočítaných podúloh po zlyhaní (0-nevynulovať, 1-vynulovať)	0
-g	pridelí si všetky voľné podúlohy bez dohody (0-dohodne sa, 1-pridelí hneď)	0
-p	počet procesorov	1
-t	počet podúloh	1
-T	μ normálneho rozdelenia pre časové zložitosti podúloh	1.0
-U	σ normálneho rozdelenia pre časové zložitosti podúloh	0.0
-P	μ normálneho rozdelenia pre μ zlyhaní každého procesora (0 = spoľahlivý procesor)	0.0
-R	σ normálneho rozdelenia pre μ zlyhaní každého procesora	0.0
-D	μ normálneho rozdelenia pre časové dĺžky zlyhaní	1.0
-E	σ normálneho rozdelenia pre časové dĺžky zlyhaní	0.0
-S	σ normálneho rozdelenia pre zlyhanie každého procesora	0.0
-l	úroveň logovacích správ (none, trace, debug, msg, info, error)	info
`lbmodel run <input-file>`		
-	cesta k súboru so vstupnými parametrami	-

lbmodel gen-input

Jediným povinným parametrom je *-o*, pri ostatných, keď sa nezadajú, tak sa použijú predvolené hodnoty. Parameter *-i* dopĺňa zo zadaného súboru parametre, nezadané pri spúšťaní programu. Taktiež parametre, ktoré sú do výstupného súboru generované (ako konkrétne hodnoty časov zlyhaní procesora, veľkosť podúloh, . . .), sa znovu vytvoria len v prípade, ak bol zmenený nejaký

parameter, z ktorého boli pôvodne generované.

lbmodel run

Má iba jeden parameter, cestu k vstupnému súboru, ktorý je povinný.

Príklad vstupného súboru

Nasledujúci vstupný súbor bol vygenerovaný príkazom:

```
lbmodel gen-input -o output -c 2 -d 0.01 -a npart -f 0 -g 0 -p
   2 -t 10 -T 1 -U 0.1 -P 2 -R 0.1 -D 1 E 0.1 -S 0.1 -l debug
```

```
LOG_LEVEL=debug                          -l log level je na úrovni debug
NR_CORES=2                               -c 2 reálne pracujúce procesory
LATENCY=0.010                            -d oneskorenie správ je 0.01s
TSKS_AMOUNT=npart                        -a počet podúloh bude n-tina
FORGET_STATUSES=0                        -f 0 = čo už raz vyrátal nezabudne
GET_IMMEDIATELY=0                        -g 0 = dohoda o nepridelenej práci
NR_PROCS=2                               -p 2 procesory
NR_TASKS=7                               -t 8 poúloh
MU_TASKS=1.00 SIG_TASKS=0.10             -T a -U μ a σ podúloh
MU_FALLS_PROB=2.00 SIG_FALLS_PROB=0.10-P a -R μ a σ pre μ zlyhaní proces.
MU_FALLS_DUR=1.00 SIG_FALLS_DUR=0.00  -D a -E μ a σ trvania zlyhaní
SIG_NEXTFALL=0.10                        -S σ zlyhania procesorov

------ FALL PROBABILITIES -----          μ zlyhania každého procesora
1.93 1.98

------------ FALLS ------------          Časové úseky, po ktorých uplynutí
P0 1.92 1.84 1.67 2.08 1.99 1.88 ...     procesor zlyhá (v sekundách). Po
P1 1.89 1.87 1.98 2.08 1.98 2.06 ...     poslednom zlyhaní prechádza na prvý.
                                         (P0 = procesor 0)
---------- DURATIONS ----------          Časové úseky, po ktorých uplynutí
P0 1.00 1.00 1.00 1.00 1.00 1.00 ...     sa procesor zobudí (v sekundách).
P1 1.00 1.00 1.00 1.00 1.00 1.00 ...     Po poslednom zobudení prechádza na
                                         prvý.
------ TASKS COMPLEXITIES -----          Veľkosti všetkých podúloh
0.74 1.09 1.20 1.05 1.04 0.96 1.04       (v sekundách)
```

Vstupné parametre nie sú špeciálne kontrolované, čo sa týka typovej kontroly, alebo či je so zadanými parametrami schopný v konečnom čase vyriešiť úlohu úspešne. Riešenie prípadných problémov je popísané v sekcii 5.3

Príklad logovacej správy

```
[inf][0][30955664][ST_WAIT4END] changeStatus:387:Some message
```

inf	úroveň logovanej správy
0	ID procesora
30955664	časová pečiatka
ST_WAIT4END	stav procesora
changeStatus	funkcia, v ktorej sa aktuálne nachádza
387	príslušný riadok v zdrojovom kóde
Some message	samotná správa

Príklad výstupného súboru

Každý procesor na záver vytvorí súbor s lokálnymi štatistikami v tvare *<input-file>_ID*. Teda zo vstupného súboru z predchádzajúceho príkladu by vznikli 2 súbory *output_0* a *output_1*.

Následne sa na koniec vstupného súboru pripíšu globálne štatistiky zozbierané z lokálnych od každého procesora. Teda v prípade predchádzajúceho príkladu by to bol súbor *output*.

```
\subsubsection{Merania na nespoľahlivých procesoroch so sekundovou dobou
výpadku}
NR_FALLS=2              Počet zlyhaní procesora
NR_INTER_MESSAGES=2     Počet výpočtov podúloh, ktoré prerušilo zlyhanie
NR_SENDS=6              Počet správ odoslaných jednému procesoru
NR_OFF_SENDS=0          Počet správ s oznámením, že je padnutý
NR_BC=5                 Počet správ odoslaných všetkým procesorom
NR_NEED_WORK_BC=0       Počet žiadostí o nové podúlohy
NR_WANT_UNASSIGN_BC=2   Počet žiadostí o nepridelené podúlohy
NR_SOLVED_TASKS=5       Počet vyriešených podúloh
EXPECTED_TIME=4.96      Najlepší možný čas riešenia podúloh
WASTE_TIME=0.52         Čas strávený riešením podúloh pred zlyhaním
WAIT4TASKS_TIME=0.76    Čas strávený čakaním na nové podúlohy
FALL_TIME=2.02          Čas stravený v padnutom stave
TOTAL_TIME=6.91         Celkový čas riešenia celej úlohy
NOT_WASTE_TIME=3.03     Čas strávený na vyriešených podúlohách
SOLVED_TASKS= 6 7 0 8 9 Zoznam vyriešených podúloh
```

5.3 Možné problémy

Program sa nespustí (vypíše len ... checking MPS Server ...)

Na príčine je pravdepodobne MPS server a pravdepodobne jedna z nasledujúcich situácií:

- MPS server môže niekedy zlyhať, vtedy je potrebné skontrolovať, či beží. Ak nie, je nutné ho znovu spustiť.
- MPS server niekedy odmietne klienta (čo aj vypíše do konzoly), potom je program nutné skúsiť opätovne spustiť. Ak naďalej klienta odmieta, postačí reštartovanie MPS Servera.

Program neustále vypisuje nejaké logovacie správy

Pravdepodobne boli zadané také parametre, pri ktorých šanca na úspešné dokončenie úlohy je veľmi malá, alebo nulová. Je potrebné zmeniť vstupné parametre (znížiť veľkosti podúloh, zväčšiť oneskorenia medzi zlyhaniami, . . .).

```
FALLS=4                 Súčet zlyhaní procesorov
INTER_MESSAGES=4        Súčet výpočtov podúloh, ktoré prerušilo zlyhanie
DUPLICATES=0            Počet duplicitne vyriešených podúloh
SENDS=10                Súčet správ odoslaných jednému procesoru
OFF_SEND=3              Súčet správ s oznámením, že je padnutý
BC=12                   Súčet správ odoslaných všetkým procesorom
NEED_WORK_BC=3          Súčet žiadostí o nové podúlohy
UNASSIGN_BC=3           Súčet žiadostí o nepridelené podúlohy
EXPECTED_TIME=10.18     Najlepší možný čas riešenia všetkých podúloh
WASTE_TIME=1.18         Súčet časov strávený riešením podúloh pred pádom
WAIT4TASKS_TIME=1.31    Súčet časov strávený čakaním na nové podúlohy
COMPUTE_TIME=5.06       Súčet časov stravený v padnutom stave
FALL_TIME=4.03          Súčet časov riešenia celej úlohy
TOTAL_TIME=13.82        Súčet časov strávený na vyriešených podúlohách
REAL_TIME=6.91          Čas najlepšieho procesora
```

Program neskončil, ale dlhú dobu nič nevypísal

Pravdepodobne boli zadané také parametre, pri ktorých dĺžka výpadkov je veľmi veľká, prípadne postačí znížiť logovaciu úroveň, aby program detailnejšie vypisoval svoju činnosť. Je teda vhodné pozmeniť vstupné parametre.

Program niekedy na konci behu vypíše „Inconsistency detected by ld.so: dl-open.c: 256: dl_open_worker: Assertion ..."

Príčina nebola zistená, každopádne nemá žiadny dopad na program ako taký.

Po manuálnej zmene parametrov vo vstupnom súbore, program padá

Vstupný súbor sa generuje programom, neodporúča sa meniť parametre priamo vo vstupnom súbore. Program si nerobí žiadnu validačnú kontrolu vstupného súboru.

5.4 Výsledky meraní

Nasledujúce pojmy nie sú z tabuliek experimentov úplne zrejmé, preto je potrebný ich bližší popis:

- Nw (Nu) = N označuje počet procesorov, w značí, že práca bola simulovaná „aktívne" (pomocou *empty_work*), u označuje, že práca bola simulovaná „pasívne" (pomocou *usleep*).
- počet zlyhaní = suma počtu zlyhaní všetkých procesorov.
- počet prerušených výpočtov = suma všetkých prerušení výpočtu podúlohy všetkých procesorov.
- počet duplicitných výpočtov = počet opakovane vyriešených podúloh
- počet správ (point-to-point) = počet všetkých správ adresovaných práve jednému procesoru
- počet správ (broadcast) = suma správ, ktoré procesory adresovali všetkým procesorom súčasne.
- teoretické trvanie výpočtu = súma časov všetkých podúloh.
- trvanie prerušených výpočtov = suma časov, ktoré procesory strávili na prerušených podúlohách.
- trvanie čakaní na podúlohy = suma časov, ktoré procesory strávili čakaním na nevyriešené podúlohy.
- trvanie zlyhaní = suma časov, ktoré procesory strávili v zlyhaní.
- trvanie výpočtu podúloh = suma časov, ktoré procesory strávili výpočtom úspešne vyriešených podúloh
- celkové trvanie výpočtu = suma časov každého procesora, od spustenia programu, až po jeho ukončenie.
- skutočný čas výpočtu = najkratší čas výpočtu s pomedzi všetkých procesorov.

Merania na spoľahlivých procesoroch

počet procesorov	**1w**	**2w**	**4w**	**1u**	**2u**	**4u**	**8u**	**16u**
počet podúloh	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
počet zlyhaní	0	0	0	0	0	0	0	0
počet prerušených výpočtov	0	0	1	0	0	5	6	10
počet duplicitných výpočtov	0	0	0	0	0	0	0	0
počet správ (point-to-point)	0	2	36	0	2	36	237	734
počet správ (broadcast)	1	4	16	1	4	16	42	69
teoretické trvanie výpočtu	30	30	30	30	30	30	30	30
trvanie prerušených výpočtov	0	0	1.10	0	0	1.92	1.70	4.17
trvanie čakaní na podúlohy	0	0.21	1.52	0	0.20	1.40	6.41	19.35
trvanie zlyhaní	0	0	0	0	0	0	0	0
trvanie výpočtu podúloh	32.69	32.69	32.69	30.03	30.04	30.04	30.02	30.01
celkové trvanie výpočtu	32.69	32.94	36.44	30.03	30.25	33.25	39.02	56.64
skutočný čas výpočtu	32.69	16:46	9.07	30.03	15.12	8.27	4.70	3.31

lbmodel gen-input -o output -p X -t 30 -T 1 -c Y

Prvá tabuľka obsahuje výsledky v situácii, keď žiadny procesor nie je nespoľahlivý. X označuje počet procesorov a Y, či sa procesor reálne vyťažuje, alebo nie. Čo je v tabuľke zaujímavé je práve porovnanie experimentov s vyťažovaním procesora a bez. Počet odoslaných správ majú identický a aj celkový čas odpočítaním času stráveného výpočtom, dáva pre rovnaký počet procesorov približne tie isté rozdiely. Z čoho vyplýva, že na simuláciu práce, výpadkov a čakania na voľné podúlohy sa dá používať aj funkcia *usleep*.

	-g 0				-g 1			
počet procesorov	**4w**	**4u**	**8u**	**16u**	**4w**	**4u**	**8u**	**16u**
počet podúloh	**30**	**30**	**30**	**30**	**150**	**150**	**150**	**150**
počet zlyhaní	0	0	0	0	0	0	0	0
počet prerušených výpočtov	5	4	6	12	3	5	5	9
počet duplicitných výpočtov	0	0	0	0	0	0	0	0
počet správ (point-to-point)	36	36	196	644	42	45	300	1829
počet správ (broadcast)	16	16	36	62	18	19	52	139
teoretické trvanie výpočtu	30	30	30	30	150	150	150	150
trvanie prerušených výpočtov	2.25	1.76	3.78	9.28	1.44	2.0	3.85	3.70
trvanie čakaní na podúlohy	1.74	1.4	3.61	14.37	2.47	1.7	5.58	27.06
trvanie zlyhaní	0	0	0	0	0	0	0	0
trvanie výpočtu podúloh	32.68	30.04	30.03	30.04	164.6	150.2	150.2	150.1
celkové trvanie výpočtu	36.89	33.25	37.34	51.53	168.9	153.7	160.5	192.1
skutočný čas výpočtu	9.17	8.28	4.58	3.09	42.19	38.41	19.91	11.6

lbmodel gen-input -o output -p X -t Y -T 1 -c Z -g 1

X je v tomto prípade počet procesorov, Y počet podúloh a Z označuje, či sa procesor reálne vyťažuje, alebo nie. Zaujímavé je v tomto prípade porovnanie, keď sa procesory nedohadujú, kto si vezme nepridelené podúlohy. Porovnaním hodnôt s predchádzajúcou tabuľkou je zrejmé, že pri tomto prístupe, keď sa procesory nedohadujú, je so zvyšujúcim sa počtom procesorov efektivita vyrovnávania zaťaženia vyššia, oproti prístupu, keď sa procesory dohadujú.

	-a one		**-a npart**		**-a all**	
počet procesorov	**4w**	**4w**	**4w**	**16u**	**16u**	**16u**
počet podúloh	**30**	**30**	**30**	**30**	**30**	**30**
počet zlyhaní	0	0	0	0	0	0
počet prerušených výpočtov	30	4	3	13	9	45
počet duplicitných výpočtov	0	0	0	0	0	1
počet správ (point-to-point)	75	36	99	644	734	2034
počet správ (broadcast)	29	16	37	69	70	157
teoretické trvanie výpočtu	30	30	30	30	30	30
trvanie prerušených výpočtov	2.28	2.23	1.94	6.16	4.75	19.15
trvanie čakaní na podúlohy	3.67	2.27	5.28	13.52	18.50	66.23
trvanie zlyhaní	0	0	0	0	0	0
trvanie výpočtu podúloh	32.71	32.72	32.64	30.00	30.00	31.08
celkové trvanie výpočtu	39.20	37.47	40.53	53.61	55.83	130.5
skutočný čas výpočtu	9.74	9.31	10.10	2.89	3.24	7.82

lbmodel gen-input -o output -p X -t 30 -T 1 -a Y -c Z

X označuje počet procesorov, Y počet podúloh posielaných na žiadosť o nevyriešené podúlohy a Z označuje, či sa procesor reálne vyťažuje, alebo nie. Pri tomto experimente bol škálovaný parameter posielania množstva nevyriešených podúloh. V prípade štyroch procesorov, čo sa týka správovej zložitosti, s pomerne veľkým odstupom je na tom najlepšie parameter nastavený na n-tinu. Zaujímavé ale je, že pri posielaní jednej podúlohy (parameter nastavený na *one*), v prípade 16-tich procesorov, je správová zložitosť menšia ako v prípade n-tiny. Tu sa ale prejavil fakt, že pomer medzi počtom podúloh a počtom procesorov je primalý. V tomto prípade ani nie 2:1, z čoho vyplýva, že n-tina pri 16-tich procesoroch a 30-tich taskoch, zaokrúhlená nadol, je 1. Posielanie všetkých (parameter *all*) nevyriešených podúloh, znamená neustále žiadosti o nové podúlohy po vypočítaní skoro každej podúlohy. Má za následok neúmerné zvýšenie správovej, aj časovej zložitosti.

Merania na nespoľahlivých procesoroch

	-g 0			**-g 1**		
počet procesorov	**2w**	**4w**	**8u**	**2w**	**4w**	**8d**
počet podúloh	**150**	**150**	**150**	**150**	**150**	**150**
počet zlyhaní	61	74	68	38	44	35
počet prerušených výpočtov	39	35	16	31	34	25
počet duplicitných výpočtov	4	0	2	14	9	28
počet správ (point-to-point)	128	478	1061	44	199	40
počet správ (broadcast)	130	166	163	48	73	68
teoretické trvanie výpočtu	15	15	15	15	15	15
trvanie prerušených výpočtov	4.32	3.96	1.6	3.48	3.86	2.50
trvanie čakaní na podúlohy	43.21	59.06	59.40	17.81	25.95	22.39
trvanie zlyhaní	7.28	8.97	6.86	4.53	5.23	3.59
trvanie výpočtu podúloh	18.47	18.05	15.33	19.65	19.7	17.90
celkové trvanie výpočtu	73.56	97.74	100.4	45.85	58.86	53.21
skutočný čas výpočtu	36.69	24.27	12.36	22.84	14.69	6.51

lbmodel gen-input -o output -p X -t 30 -T 0.1 -P 1 -S 0.5 -D 0.1 -d 0.2 -c Z

X označuje počet procesorov a Z označuje, či sa procesor reálne vyťažuje, alebo nie. Experiment spočíval v tom, že či duplicitné výpočty výrazne prispievajú do výsledného času výpočtu, vzhľadom na prácu navyše s tým spojenú. Z tabuľky vyplýva, že efektivita prístupu, s čo najväčšou minimalizáciou duplicitných výpočtov, má za následok neúmerne veľa odoslaných správ. A keďže je nastavený aj parameter oneskorenia (-d), tak vo výslednom čase výpočtu to vytvára v niektorých prípadoch aj dva krát väčšiu prácu navyše. Parameter oneskorenia, ale výrazne ovplyvňuje aj počet duplicitných správ. Bez oneskorenia je totiž časový rozdiel medzi poslaním žiadostí o prácu a prevzatím nepridelených podúloh priveľmi malý, takže nepridelené poúlohy si vezme vo väčšine prípadov len jeden procesor, čo potvrdzuje aj nasledujúci experiment.

	-g 0	**-g 1**
počet procesorov	**8d**	**8d**
počet podúloh	**150**	**150**
počet zlyhaní	286	49
počet prerušených výpočtov	76	49
počet duplicitných výpočtov	0	2
počet správ (point-to-point)	4395	698
počet správ (broadcast)	638	108
teoretické trvanie výpočtu	30	30
trvanie prerušených výpočtov	10.65	7.7
trvanie čakaní na podúlohy	272.2	18.11
trvanie zlyhaní	286.8	48.40
trvanie výpočtu podúloh	30.10	30.54
celkové trvanie výpočtu	649.1	109.1
skutočný čas výpočtu	80.92	13.57

lbmodel gen-input -o output -p 8 -t 150 -T 0.2 -P 1 -S 0.1 -c 0 -D 1 -g X

X označuje počet procesorov a Z označuje, či sa procesor reálne vyťažuje, alebo nie. V prípade, že oneskorenie je nastavené na nulu, tak aj v prípade, že zlyhania sú časté procesory, si zriedka pridelia naraz tú istú nepridelenú úlohu. Dokazuje to aj výsledok s ôsmimi procesormi, kde síce nastalo čí zlyhaní, ale len 2 duplicitné výpočty. Naproti tomu, pri prístupe, kde sa dohaduje, kto si pridelí nepridelené podúlohy je 6 krát viac správ odoslaných. Na výslednom čase sa to prejavilo štyri krát horším časom.

5.5 Záver meraní

Aj po maximálnej snahe o zredukovanie duplicitných výpočtov, čas paralelného výpočtu nie je podľa očakávaní, aj keď si odpočítame čas, ktorý strávil v nečinnosti. V najväčšej miere sa to ukazuje pri väčšom počte výpadkov, kedy pomerne často vznikajú nepridelené úlohy a dohoda medzi procesormi, kto si ju vezme má istú réžiu, ktorá sa tu v dosť značnej miere prejavila, čo dokumentujú to aj posledné dva experimenty. Posielanie správ je príliš drahé a minimalizácia duplicitných výpočtov tú cenu nevykompenzuje. Každopádne model je použiteľný v prostredí, kde má vyššiu prioritu spoľahlivosť ako čas výpočtu.

Kapitola 6

Záver

Cieľom diplomovej práce bolo navrhnúť model vyrovnávania zaťaženia na sieti pozostávajúcej len z nespoľahlivých procesorov, následne ho implementovať a škálovaním jeho atribútov pri experimentoch zhodnotiť jeho efektivitu paralelného výpočtu. Model bol navrhnutý a aj sa ho podarilo úspešne implementovať. Dôraz bol kladený hlavne na splnenie podmienok, ktoré boli stanovené pred začiatkom samotného návrhu, a tiež kvôli efektivite paralelného výpočtu a na minimalizáciu duplicitných výpočtov. Z hľadiska efektivity paralelného výpočtu, ale nie je tak efektívny ako sme očakávali, čo ukázali aj experimenty. No ako bolo v závere experimentov spomínané, určite by mal v istých prípadoch uplatnenie. Určite to teda nie je zlý model, už len z toho pohľadu, do akej miery je tolerantný voči chybám. Na záver môžeme konštatovať, že cieľ sa nám podarilo dosiahnuť.

Obsah CD

K práci je priložené CD so zdrojovými kódmi, skompilovaná verzia pre Linux (kompilátorom gcc) a elektronická verzia tejto práce.

Adresárová štruktúra a popis:

/bin/ - skompilovaná verzia programu
/src/ - zdrojové súbory k modelu a k pomocným programom
/mps/ - MPS server (bez zdrojových kódov)
/text/ - elektronická verzia tejto práce

Literatúra

[ALR01] A. Avizienis, J. Laprie, and B. Randell. **Fundamental Concepts of Dependability**, 2001.

[BFG01] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. **The Natural Work-Stealing Algorithm is Stable**. In *IEEE Symposium on Foundations of Computer Science*, pages 178–187, 2001.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. **Cilk: An Efficient Multithreaded Runtime System**. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996. citeseer.ist.psu.edu/blumofe95cilk.html.

[CFG+05] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. **Fault tolerant high performance computing by a coding approach**. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[Gar99] Felix C. Gartner. **Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments**. *ACM Computing Surveys*, 31(1):1–26, 1999.

[GL02] W. Gropp and E. Lusk. **Fault Tolerance in MPI Programs**. Mathematics and Computer Science Division Argonne National Laboratory Argonne, IL 60439, USA, 2002.

[Hag97] Torben Hagerup. **Allocating Independent Tasks to Parallel Processors: An Experimental Study**. *J. Parallel Distrib. Comput.*, 47(2):185–197, 1997.

[Jeo96] Karpjoo Jeong. ***Fault-tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions***. PhD thesis, New York University, 1996. citeseer.ist.psu.edu/jeong96faulttolerant.html.

[JGKR05] Samir Jafar, Thierry Gautier, Axel W. Krings, and Jean-Louis Roch. **A Checkpoint/Recovery Model for Heterogeneous Dataflow Computations Using Work-Stealing**. In LNCS Springer-Verlag, editor, *EUROPAR'2005*, Lisboa, Portogal, August 2005.

[LAK92] J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. ***Dependability: Basic Concepts and Terminology***. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[Pau07] Roman Pauer. **Centralizovaný systém pre výmenu správ medzi paralelnými procesmi**. Master's thesis, Univerzita Komenského, Fakulta matematiky, fyziky a informatiky, Bratislava, 2007.

[Pla03] Tomas Platchetka. ***Event-Driven Message Passing and Parallel Simulation of Global Illumination***. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2003.

[Pla04] T. Plachetka. **Tuning of Algorithms for Independent Task Placement in the Context of Demand-Driven Parallel Ray Tracing**. In *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Parallel Graphics and Visualisation, Eurographics Proceedings Series, Eurographics/ACM SIGGRAPH, Grenoble, France.*, 2004.

[RW02] T. Raghavan and N. R. S. Waghmare. **DPAC: an object-oriented distributed and parallel computing framework for manufacturing applications**. *IEEE Transactions on Robotics and Automation*, 18(4):431–443, August 2002.

[TT85] Asser N. Tantawi and Don Towsley. **Optimal static load balancing in distributed computer systems**. *J. ACM*, 32(2):445–465, 1985.

[vNKB01] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. **Efficient Load Balancing for Wide-area Divide-and-Conquer Applications**. In *Proc. Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 34–43, Snowbird, UT, June 2001.

[Zam05] Simon Zamecnik. **Dynamický load balancing pre PVM s využitím migrácie úloh**. Master's thesis, Univerzita Komenského, Fakulta matematiky, fyziky a informatiky, Bratislava, 2005.