Univerzita Mateja Bela v Banskej Bystrici

Fakulta prírodných vied

Optimalizácia plánovania úloh v gridoch a cloudoch pomocou GPGPU

diplomová práca

066b8d5c-2dea-487f-8b68-8e463c4d692c

2014 Bc. Michal Povinský

Univerzita Mateja Bela v Banskej Bystrici

Fakulta prírodných vied

Optimalizácia plánovania úloh v gridoch a cloudoch pomocou GPGPU

diplomová práca

066b8d5c-2dea-487f-8b68-8e463c4d692c

Študijný program:	Aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	9.2.9. aplikovaná informatika
Katedra:	KIN FPV – Katedra informatiky
Vedúci diplomovej práce:	Doc. Ing. Jarmila Škrinárová, PhD.

Banská Bystrica, 2014 Bc. Michal Povinský

Univerzita Mateja Bela v Banskej Bystrici
Fakulta prírodných vied

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Michal Povinský
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.9. aplikovaná informatika
Typ záverečnej práce: Magisterská záverečná práca
Jazyk záverečnej práce: slovenský

Názov: Optimalizácia plánovania úloh v gridoch a cloudoch pomocou GPGPU.

Anotácia: Analyzujte modely plánovania úloh v gridoch a cloudoch. Navrhnite, implementujte a overte model optimalizácie plánovania úloh v gridoch a cloudoch pomocou GPGPU.

Vedúci: doc. Ing. Jarmila Škrinárová, PhD.
Katedra: KIN FPV - Katedra informatiky

Dátum zadania: 10.12.2012

Dátum schválenia: 08.04.2013

prof. Ing. Miroslav Svítek, Dr.
garant študijného programu

Abstrakt

Michal Povinský: *Optimalizácia plánovania úloh v gridoch a cloudoch pomocou GPGPU.* [Diplomová práca]. Univerzita Mateja Bela v Banskej Bystrici, Fakulta prírodných vied, katedra informatiky. Vedúci: Doc. Ing. Jarmila Škrinárová, PhD., 2014. 94 s.

Efektívne plánovanie úloh v kontexte gridového počítania prináša zložitý problém často riešený zjednodušeným technikami. Optimálne riešenie nie je dosiahnuteľné, pretože je to NP-úplný problém. Táto práca sa zameriava na návrh pokročilých plánovacích techník používajúcich GPGPU a simulované žíhanie použiteľných na gridové plánovanie s lepším výkonom ako často používané techniky na báze radu úloh.

Kľúčové slová: plánovanie úloh, simulátor, algoritmus

Abstract

Michal Povinský: *Job scheduling optimization in grids and clouds using GPGPU.* [Diploma thesis]. Matej Bel University in Banská Bystrica, Faculty of natural sciences, department of computer science. Supervisor: Doc. Ing. Jarmila Škrinárová, PhD., 2014. 94pgs.

Effective job scheduling in the context of Grid computing introduces complex problem often solved by simplified techniques. Optimal solution is not achievable since it is a NP-complete problem. This work concentrates on the design of advanced scheduling techniques using GPGPU and simulated annealing usable for Grid scheduling with better performance than frequently used queue-based techniques.

Keywords: job scheduling, simulator, algorithm

Predhovor

Efektívne plánovanie v prostredí gridu predstavuje komplexný problém, ktorý dodnes nie je uspokojivo vyriešený. Nájsť optimálny rozvrh, to znamená priradiť úlohy v čase na dostupné zdroje predstavuje NP-úplný problém, ktorý je pre väčší počet úloh v rozumnom čase neriešiteľný. Ako vhodné sa preto javí hľadanie suboptimálnych riešení (približujúcich sa k optimu), pre ktoré existujú rýchlejšie algoritmy. Použitím pokročilých plánovacích techník, ako sú napríklad heuristické metódy optimalizácie globálneho rozvrhu vzniká priestor pre významné skvalitnenie tohto rozvrhu skrátením času dokončenia úloh a optimalizáciou využitia gridu.

Všetky výpočtové úlohy sa dajú riešiť na bežnom procesore. Grafické karty sa orientujú predovšetkým na jednoduchšie operácie a ich paralelné spracovanie. Vďaka optimalizácii to potom zvládnu mnohonásobne rýchlejšie. Procesor sa síce často považuje za mozog počítača, ale napriek tomu je to veľmi často oveľa jednoduchšie zariadenie ako grafická karta. Ideálnym príkladom nasadenia všeobecných výpočtov na grafických procesoroch (GPGPU) je spracovanie jednej úlohy nad väčším množstvom dát. GPU rozdelí úlohu medzi svoje výpočtové jednotky a vykoná celú operáciu naraz, alebo len v niekoľkých cykloch. Podobne rýchlo sa dá pracovať aj s vektormi, maticami, či inými vstupmi s pomerne jednoduchými operáciami, alebo rozsiahlymi dátami. Grafická karta preto nemusí slúžiť len na hry, nasadením GPGPU sa z nej stal silný viacúčelový nástroj využiteľný pre rozvrhovanie úloh v gridoch a cloudoch.

Moja veľká vďaka patrí pani Doc. Ing. Jarmile Škrinárovej PhD., že ma inšpirovala k spracovaniu témy optimalizácie plánovania úloh v gridoch a cloudoch pomocou GPGPU. Svojou podporou, cennými radami a pripomienkami mi pomohla hlbšie preniknúť do fascinujúceho sveta GPGPU a otvoriť perspektívu dosiahnutia cieľa ďalšieho zrýchľovania riešenia zložitých problémov optimalizáciou a paralelizáciou na viacerých grafických kartách a počítačoch.

Poďakovanie

Ďakujem vedúcej diplomovej práce Doc. Ing. Jarmile Škrinárovej PhD. za cenné rady a odbornú pomoc, ktoré mi poskytla pri jej vypracovaní.

Obsah

Zoznam skratiek

ASJS	Adaptive scoring job scheduling – plánovanie úloh s adaptívnym hodnotením
CPU	Central processing unit – centrálny procesor
CUDA	Compute unified device architecture – prostredie vytvorené spoločnosťou NVIDIA pre výpočty na grafickej karte
FSA	Fast simulated annealing – rýchle simulované žíhanie
GPGPU	General-purpose computing on graphics processing units – všeobecné výpočty na grafických procesoroch
GPU	Graphics processing unit – grafický procesor
OpenCL	Open computing language – univerzálne prostredie pre výpočty na grafickej karte
SA	Simulated annealing – simulované žíhanie
SIMT	Single instruction multiple threads – architektúra – jedna inštrukcia, viac vlákien

Zoznam obrázkov

Úvod

Plánovanie sa využíva vo všetkých odvetviach ľudskej činnosti. Nezaobídeme sa bez neho v školách pri tvorbe rozvrhov, v zdravotníctve pri plánovaní operácií, v doprave pri tvorbe cestovných poriadkov, letových, plavebných plánov, v priemysle pri optimalizácii využívania kapacity strojov, v stavebníctve za účelom dodržania časového harmonogramu výstavby a dodržania rozpočtu a pod. Plánovanie je rozhodovací proces, spočíva vo formulácii cieľov a ciest, ktorými sa ciele majú dosiahnuť a to čo najefektívnejšie. Plánovanie sa zaoberá alokáciou zdrojov s cieľom optimalizácie na základe jedného alebo viacerých kritérií. Zdroje aj úlohy môžu mať rôznu podobu. Zdrojmi môžu byť stroje v podniku, pristávacie plochy letiska, procesory vo výpočtovom stredisku. Úlohami môžu byť pracovné operácie technologického postupu, pristávanie a vzlet lietadiel, beh počítačových programov. Každá z úloh má rôznu prioritu, požadovaný čas ukončenia, čas začatia atď. Ciele môžu byť tiež rôznorodé, napr. maximalizácia počtu dokončených úloh v určitom čase, minimalizácia času dokončenia poslednej úlohy a pod.

Pritom aktuálne problémy v živote, vo vede a inžinierstve sa so sústavným pokrokom ľudskej kultúry stávajú stále komplikovanejšími a na ich analýzu a riešenie je potrebný čoraz väčší výpočtový výkon. Superpočítač už nie jedinou možnosťou riešenia komplexných problémov. Spôsobilo to najmä zrýchlenie osobných počítačov a sietí. Gridová technológia spájajúca klastre pomocou vysokorýchlostných sietí môže dosiahnuť rovnaký výkon ako superpočítač, ale za podstatne nižšiu cenu, pretože na rozdiel od iných vysoko výkonných riešení je grid zložený z bežných osobných počítačov s nízkou obstarávacou cenou. Počítače nemusia mať rovnakú architektúru, konfiguráciu, výkon ani softvérové vybavenie a môžu byť rozmiestnené na rozličných miestach. Grid je preto heterogénne dynamické výpočtové prostredie.

Distribuované počítanie je charakteristické používaním spoločných hardvérových a softvérových prostriedkov. Paralelné počítanie podporuje výpočtový výkon. Cieľom gridového počítania je využitie sily tak distribuovaného ako aj paralelného počítania. Tento cieľ sa pri gridovom počítaní dosahuje zhromažďovaním nevyužitých prostriedkov na internete. Ide o to, aby sa nevyužité procesorové cykly a nevyužitý úložný priestor mohol využiť. Grid sa môže chápať ako rozsiahly distribuovaný systém organizačne samostatných prvkov, ktorými môžu byť individuálne počítače (pamäte, pevné disky) a informačné systémy vzájomne pospájané počítačovou sieťou.

Plánovanie úloh v prostredí gridu vo všeobecnosti znamená alokáciu úloh na dostupné zdroje v čase tak, aby boli splnené zadané kritériá (napr. minimalizácia doby dokončenia poslednej úlohy, maximalizácia využitia zdrojov). Keďže grid je heterogénny systém, plánovanie nezávislých úloh na ňom je komplikované. Efektívne plánovanie v tomto prostredí predstavuje komplexný problém, novú výzvu, ktorý nie je dodnes uspokojivo vyriešený. Aby bol výkon gridu úplne využitý, je potrebný efektívny algoritmus plánovania úloh na priraďovanie úloh prostriedkom. Nájdenie optimálneho algoritmu, t.j. priradenie úloh v čase na dostupné zdroje predstavuje NP-úplný problém, ktorý je pre veľký počet úloh v rozumnom čase neriešiteľný. Algoritmus je možné významne skvalitniť (skrátenie času dokončenia úloh, zlepšenie priepustnosti systému a optimalizácia využitia gridu) nasadením pokročilých plánovacích techník, ako sú napr. heuristiky. [4]

Cieľom diplomovej práce je navrhnúť, implementovať a overiť model optimalizácie plánovania úloh v gridoch a cloudoch. Keďže proces optimalizácie je náročný na výpočtový výkon, je vhodné využiť pri výpočte grafické karty a otestovať zrýchlenie, ktoré sa dá ich použitím dosiahnuť. Donedávna sa grafické karty využívali len na zábavu a počítačové hry. Vznik aplikačného rozhrania na všeobecné výpočty na grafických kartách priniesol programátorom možnosti využívať masívny paralelný výkon moderných grafických kariet na urýchľovanie všeobecných výpočtov.

V druhej časti práce je opísaná problematika plánovania a rozvrhovania a teoretické východiská pre efektívne plánovanie. Tretia časť je venovaná analýze modelov plánovania úloh a rozboru optimalizačných algoritmov. V štvrtej časti venovanej samotnému návrhu modelu optimalizácie plánovania úloh je opísaná funkcionalita

a možnosti simulátora plánovania úloh a navrhnutý model optimalizácie a jeho implementácia a overenie. Metodika a výsledky experimentov, ktoré overili funkčnosť, správnosť a možnosti zvoleného riešenia sú uvedené v piatej časti.

Kapitola 1

Gridové počítanie

1.1 História gridových systémov

Gridy sa objavili na počiatku distribuovaného klastrového počítania. Klaster znamená také spojenie dvoch a viacerých počítačov z hľadiska softvéru aj hardvéru, že sa správajú ako jeden počítač. Takéto riešenie prinieslo možnosť paralelného spracovania výpočtových úloh a vyrovnávanie záťaže individuálnych výpočtových zdrojov. Gridové počítanie má pôvod v distribuovanom klastrovom riešení a pozdvihlo ho na kvalitatívne vyššiu úroveň.

Pojem grid sa prvýkrát objavil v roku 1997, kedy ho Ian Foster a Carl Kesselman použili na označenie novej infraštruktúry pre vedu 21. storočia. Grid je ďalšou evolučnou fázou vývoja distribuovaného počítania. V gride môžu byť prepojené geograficky vzdialené klastre tak, aby poskytovali veľkú výpočtovú silu a veľa úložného priestoru. Výpočtovými elementami v gride môžu byť aj individuálne superpočítače. Gridové technológie takto umožňujú prístup k výpočtovým a pamäťovým prostriedkom po celom svete. Grid sa tak stáva globálnou sieťou počítačov reprezentujúcou obrovské výpočtové prostredie poskytujúcou bezpečné a vysokovýkonné metódy prístupu k vzdialeným prostriedkom. Jeho prostriedky majú možnosť spoločne využívať rôzne inštitúcie i jednotlivci. [3]

1.2 Definícia gridu a charakteristika gridového prostredia

Podľa [3] je grid rozsiahly distribuovaný systém organizačne samostatných elementov, ktorými môžu byť individuálne počítače (pamäte, pevné disky) a informačné systémy vzájomne prepojené počítačovou sieťou. Grid je možné vnímať podobne ako internet, ktorý spája milióny počítačov na celom svete, umožňuje komunikáciu a sprístupňuje informácie na webových stránkach. Podobne aj grid umožňuje užívateľom využívanie výpočtovej sily a diskovej kapacity nevyhnutnej pre beh výpočtovo náročných aplikácií.

Jedna z prvých definícií gridu bola použitá v [7]: „Výpočtový grid je hardvérová a softvérová infraštruktúra, ktorá poskytuje spoľahlivý, štandardizovaný, všadeprítomný a lacný prístup ku špičkovým výpočtovým službám“.

Neskôr bola táto definícia spresnená o kľúčové vlastnosti, ktorými musí prostredie, aby sa stalo gridom disponovať. Podľa nej grid:

- koordinuje zdroje nepodliehajúce centralizovanej správe,
- používa štandardné, otvorené, všeobecné protokoly a rozhrania,
- poskytuje netriviálnu kvalitu a kvantitu služieb.

Napriek tomu, že gridy majú pôvod v distribuovanom počítaní, nemôžeme ich interpretovať rovnakým spôsobom. Distribuované prostredie sa od gridov líši tým, že distribuované aplikácie predstavujú špecializované systémy pre určitý cieľ alebo skupinu užívateľov a gridy sú univerzálnou platformou. Gridy sú síce postavené na distribuovanom prostredí, ale prinášajú kvalitatívne úplne nové vlastnosti [3]:

- rôzne druhy zdrojov,
- striktne sa nevyžaduje rovnaký hardvér, dáta a aplikácie,
- rôzne druhy interakcií,
- rôzne užívateľské skupiny a aplikácie interagujú s gridom rôzne,

- dynamická povaha (zdroje a užívatelia často pribúdajú, ubúdajú, menia sa).

Gridové prostredie je charakteristické veľkou početnosťou výpočtových zdrojov, ktoré sú:

- heterogénne,
- geograficky oddelené,
- spojené heterogénnymi sieťami,
- sú úplne pod kontrolou svojich vlastníkov.

Pre prístup k jednotlivým zdrojom sú potrebné rôzne bezpečnostné opatrenia a pravidlá správy zdrojov, čím sa zabezpečí unifikovaný bezpečný prístup užívateľov k prostriedkom, ktoré práve potrebujú, spoľahlivé dodávky žiadaných služieb a správne vyúčtovanie za ich spotrebu.

1.3 Typy gridov

1.3.1 Výpočtový grid

Výpočtový grid je grid poskytujúci výpočtový servis, v rámci ktorého poskytuje zabezpečené služby pre spúšťanie aplikácií na distribuovaných výpočtových zdrojoch. V podstate to je virtuálny superpočítač, ktorý dynamicky spája výpočtové kapacity individuálnych počítačov za účelom poskytnutia platformy pre riešenie náročných aplikácií neriešiteľných pomocou jediného systému.

1.3.2 Dátový grid

Dátový grid znamená spracovanie veľkých objemov dát pomocou služieb výpočtového gridu. Zdieľa veľké množstvo dát, poskytuje zebezpečený prístup k týmto dátam a umožňuje ich následné spravovanie. Je to riešené formou replikovaných dátových katalógov, ktoré vytvárajú ilúziu jednotného hromadného dátového úložišťa.

1.3.3 Informačný grid

Informačný grid (znalostný, aplikačný) sa snaží o rozšírenie možností dátových gridov o poskytovanie kategorizácie dát, zdieľanie znalostí a pod. Jeho súčasťou je virtuálne prostredie pre spoluprácu alebo virtuálne laboratóriá, ktoré umožňujú vzdialenú kontrolu a správu vybavenia senzorov a zariadení.

1.4 Gridové aplikácie

Gridy sú vhodné pre paralelné spracovanie úloh. Ide o úlohy, ktorých spracovanie na jednom procesore by trvalo veľmi dlho, alebo ich nároky na pamäť sú príliš veľké. Aplikačne sú gridy vhodné pre spracovanie veľkého množstva malých úloh, ktoré sú navzájom nezávislé, ale tiež aj pre spracovanie rozsiahlych dátových súborov. Okrem riešenia extrémne výpočtovo náročných úloh gridové prostredie rieši aj problém dátových úložísk, umožňuje prístup k špeciálnemu hardvéru a hlavne zabezpečuje sprostredkovanie efektívneho využitia dostupných zdrojov. Pre ilustráciu počtu spracovávaných úloh na gride je na obr. 1.1 uvedené monitorovanie aktuálne bežiacich úloh na gride NorduGrid.

Okrem nasadenia gridových riešení v komerčnej sfére na riešenie dátovo a výpočtovo náročných operácií pri spracovaní rôznych komplexných investičných a finančných analýz sú ďalšie sféry, v ktorých je možné ich využitie, napr. v chémii a biológii pre analýzu a modelovanie chemických a biologických informácií, vo fyzike, lekárskych vedách pri rôznych interaktívnych simuláciách a analýzach, astronómii pri analýzach dát z teleskopov, v oblasti životného prostredia pri monitorovaní znečisťovania prostredia, predpovedi počasia, pri ochrane obyvateľstva, v multimédiách a v mnohých ďalších oblastiach života.

ARC Grid Monitor

2014-04-22 CEST 22:18:47

Processes: Grid Local

Country	Site	CPUs	Load (processes: Grid+local)	Queueing
China	BOINC Cluster	24	0+0	0+0
Denmark	Steno Tier 1 (DCSC/KU)	5208	832+2852	581+0
	Steno Tier 3 (DCSC/KU)	5208	0+3684	0+0
Finland	Aesyle (FGI)	72	8+0	1+0
	Alcyone (CMS)	892	99+618	845+0
	Alcyone (FGI)	892	82+637	0+0
	Asterope (FGI)	192	96+0	0+0
	Celaeno (FGI)	448	298+28	1+0
	DII HEP (CMS)	200	0+0	0+0
	Electra (FGI)	672	156+457	2+0
	Jade (HIP)	768	201+543	29+48
	Maia (FGI)	768	136+624	5+6
	Merope (FGI)	1612	20+1122	3896+3
	Pleione (FGI)	288	134+0	4+0
	Taygeta (FGI)	360	13+295	73+0
	Triton (FGI)	6972	12+0	1+0
	Usva (CSC/FGI/test)	144	0+0	0+0
Germany	LRZ-C2PAP	3872	1628+16	148+0
	LRZ-LMU	800	536+200	112+1
	LRZ-LMU lcg-lrz-ce0	1468	303+660	216+103
	LRZ-LMU lcg-lrz-ce3	1468	660+303	103+216
	RZG ATLAS HYDRA	167848	0+152614	0+5
	wuppertalprod	3320	2247+1357	22+13517
Hungary	Debrecen SC	1536	0+1503 (no queue info)	0+0
	NIIFI SC	768	0+538	0+11
Latvia	IMCSUL	1	0+0	0+0
Lithuania	VU-MIF-LCG2	1224	0+218	0+0
	VU-MIF-TEST	16	0+0	0+0
Norway	Abel C1(UiO/USIT)	10872	31+6637	2+6
	Abel C2(UiO/USIT)	10872	0+6652	0+6
	Abel C3(UiO/USIT)	10872	1104+5525	1+5
	fimm (BCCS/UiB)	752	0+0	0+0
Slovenia	Arctur-1	348	0+0 (queue inactive)	0+0
	Arnes	2340	2088+0	168+0
	CIPKeBiP	984	2+0	0+0
	SiGNET	2834	1524+200	42+203
	SiGNET C	2834	0+1723	0+243
	UNG	112	0+0 (queue inactive)	0+0

Obr. 1.1: Grid monitor

Kapitola 2

Plánovanie

2.1 Základné pojmy

Špecifickou črtou konania človeka je skutočnosť, že sa v predstihu zamýšľa nad úkonmi, ktoré chce realizovať. Čím sú tieto činnosti rozsiahlejšie, tým si vyžadujú viac času na analýzu pred ich samotným vykonávaním. Ľudia sa preto snažia podrobne a čo najpresnejšie odhadovať priebeh danej činnosti, vytvárajú plán. Systematický prístup k tvorbe plánov sa nazýva plánovaním. Plánovať (z latinského slova planta – náčrt budovy) znamená načrtnúť nejakú schému, vyprojektovať ako niečo vykonať. Plánovanie je rozumovou zložkou konania, prebieha v čase a týka sa budúcich udalostí. Jeho úlohou je vybrať a usporiadať úlohy tak, aby sa čo najlepšie dosiahol vytýčený cieľ, ktorým je väčšinou snaha o minimalizáciu nákladov a maximalizáciu zisku. Ziskom pritom nemusí byť vždy len finančný profit, ale aj skvalitnenie produkcie, zlepšenie pracovného prostredia, zlepšenie životného prostredia, lepšie využitie surovín, pracovných prostriedkov, informácií a pod. Pri minimalizácii nákladov ide o náklady na energiu, ľudské zdroje, časové a finančné náklady. Snaha minimalizovať jeden druh nákladov často pôsobí opačne na iný druh nákladov. Tieto protichodné tendencie musia byť pri plánovaní zohľadňované, aby výsledok plánovania, ktorým je plán – rozvrh sa čo najviac blížil k optimálnemu riešeniu.

Rozvrhovanie sa potom stará o optimálnu realizáciu plánov v prostredí s obmedzenými zdrojmi a časom. Nástrojom, pomocou ktorého sa v tomto prostredí priraďujú úlohy k systémovým prostriedkom je plánovač úloh (angl. „scheduler“).

S procesmi plánovania a rozvrhovania sa stretávame v každej oblasti ľudskej

činnosti. V praxi dochádza často k zamieňaniu pojmov plánovanie a rozvrhovanie, vo všeobecnosti však môžeme základné pojmy definovať nasledovne [9]:

- Úloha je objekt, ktorý plánujeme. Má svoje vlastnosti a vnútornú štruktúru. Skladá sa z jednej alebo viacerých operácií, ktoré môžu byť vykonávané na jednom alebo viacerých zdrojoch.
- Operácia alebo podúloha je časťou celej úlohy.
- Zdroj je prostriedkom na realizáciu operácie a spracováva jednotlivé operácie. Zdroje sú v systéme obmedzené.
- Plánovanie je dlhodobý proces vytvárania množiny vhodných aktivít tak, aby boli dosiahnuté stanovené ciele.
- Rozvrhovanie je alokácia zdrojov umiestnených v časopriestore za daných podmienok na objekty (úlohy) tak, aby boli splnené zadané kritériá, napr. minimalizácia celkovej ceny daných zdrojov s dôrazom na usporiadanie objektov v čase.
- Rozvrh je výstupom procesu rozvrhovania. Je to dátová štruktúra, ktorá obsahuje informácie o umiestnení objektov (úloh) v čase na zdrojoch.

Princíp plánovania úloh je pomerne jednoduchý. Ak sa v systéme nachádzajú úlohy, aspoň jedna z nich sa musí vykonávať. Ak je viac úloh ako procesorov, nemôžu sa vykonávať všetky naraz, niektoré sa musia pozastaviť, aby mohli byť vykonané ostatné úlohy. O tom, ktoré úlohy sa práve vykonávajú a ktoré sú pozastavené rozhoduje plánovač úloh.

2.2 Statické a dynamické plánovanie

2.2.1 Statické plánovanie

Pri statickom plánovaní a rozvrhovaní na začiatku tvorby rozvrhu poznáme všetky úlohy, ktoré majú byť narozvrhované, pričom počas plánovania nepribúdajú žiadne ďalšie požiadavky ani obmedzenia. Tiež sú na začiatku známe všetky dostupné zdroje,

t.j. máme kompletné informácie o úlohách, ktoré treba naplánovať a všetky informácie o dostupných zdrojoch. Ide o off-line plánovanie, čo znamená proces tvorby rozvrhu pred štartom systému, čiže pred tým ako sa začne spracovanie úloh na zdrojoch [9].

2.2.2 Dynamické plánovanie

Na rozdiel od statického plánovania pri dynamickom plánovaní pred začiatkom tvorby rozvrhu nie sú známe všetky úlohy, obmedzenia ani zdroje. Ide tu o zohľadnenie skutočnosti, že v reálnych aplikáciách len málokedy poznáme vopred úlohy, ich dostupnosť a tiež nepoznáme na začiatku plánovania dostupnosť zdrojov. Preto pri dynamickom plánovaní ide o nájdenie rozvrhu v podmienkach, keď dynamicky pribúdajú úlohy v dynamicky sa meniacom prostredí. Okrem pribúdania môže dochádzať aj k ubúdaniu úloh z dôvodu ich straty alebo zneplatnenia. To isté platí aj o zdrojoch, ktoré môžu pribudnúť, alebo vypadnúť zo systému. To si vyžaduje modifikáciu systému. V tomto prípade ide o on-line plánovanie [9]. Je to proces tvorby rozvrhu počas vykonávania úloh na zdrojoch, počas behu systému, teda v reálnom čase.

2.3 Plánovač úloh

Plánovač plní funkciu vrstvy medzi užívateľom a distribuovanými zdrojmi. Je to softvérový nástroj fungujúci ako rozhranie pre užívateľa gridu, ktoré ho oddeľuje od zložitosti a komplexnosti gridového prostredia a zabezpečuje mu transparentný prístup ku gridovým zdrojom. Jeho úlohou je tiež sprostredkovať pre užívateľa vyhľadávanie dostupných zdrojov, zisťovanie cien za spracovanie úloh a transport dát na jednotlivé distribuované zdroje.

Úlohy v gridovom prostredí sú najčastejšie spracovávané v dávkach. Užívateľ popíše požiadavky svojej úlohy a tento popis odovzdá plánovaču. Plánovač je zodpovedný za výber výpočtového zdroja, ktorý najlepšie vyhovuje zadaným požiadavkám. Pre obmedzené zdroje nie je možné vždy vykonávať zadané úlohy okamžite, preto musí plánovač plánovať spracovanie vzájomne previazaných úloh tak, aby bolo možné úlohy dokončiť v minimálnom čase. Plánovač teda rozhoduje nielen o tom, kde sa bude

ktorá úloha vykonávať a výsledné dáta ukladať, ale aj o tom kedy. Súčasne musí zohľadňovať aj priority jednotlivých úloh, avšak v gridových aplikáciách je najvyššou prioritou čas vykonania úlohy a termín jej ukončenia.

Z hľadiska logického zapojenia do topológie gridu sa plánovače delia na centralizované, decentralizované a hierarchické, ktoré sa odlišujú v spôsobe práce s úlohami:

- centralizovaný (lokálny) plánovač v gride býva jeden a je spoločný pre všetkých užívateľov, je typický napr. pre klastre
  - zhromažďuje informácie o stave zdrojov, optimalizuje plánovanie úloh, hlavne vyťaženie zdrojov, optimalizuje výpočty pre privilegovaných užívateľov
  - jeho výhodou je rýchlosť, lebo nemusí deliť úlohy na podúlohy a posielať úlohy na iný plánovač
  - nevýhodou je, že v prípade havárie dochádza k strate všetkých informácií o stave naplánovania jednotlivých úloh, preto musí byť dobre zabezpečený a zálohovaný a môže tiež brzdiť výkon gridu v prípade, že nie je dostatočne dimenzovaný ak je počet úloh a užívateľov vysoký
- decentralizovaný plánovač je typický pre globálne gridy
  - využíva sa pritom viac plánovačov, plánovač rozvrhuje úlohy, ak nie je schopný úlohu splniť, postúpi plánovanie úlohy na iný plánovač, alebo rozdelí úlohy na podúlohy a tie sa snaží naplánovať
  - nevýhodou je, že nedáva záruky rýchlosti a tiež rôznorodosť priorít lokálnych plánovačov (cieľ poskytovateľa) a plánovačov vyšších úrovní (požiadavky koncového užívateľa najmä čas dokončenia úlohy)
- hierarchický plánovač znamená existenciu hierarchickej štruktúry plánovačov v globálnom gride
  - slúži pre špecializované gridy na spoluprácu rôznych inštitúcií, ktoré majú svoje lokálne plánovače prepojené s nadriadenými plánovačmi vo virtuálnej

organizácii, čím sa zabezpečuje vzájomná komunikácia pri plánovaní

- výhodou je známa štruktúra a stabilná topológia
- nevýhodou je potreba transportu a delenia úloh na podúlohy ak lokálny plánovač nie je schopný úlohy rozvrhnúť.

Plánovač úloh využíva optimalizačný algoritmus. Jeho úlohou je nájdenie optima cieľovej funkcie, resp. priblížiť sa k jej optimu. V závislosti od problému môže ísť o hľadanie minima alebo maxima optimalizovanej funkcie. Preto výber optimalizačného algoritmu je závislý od cieľov, ktoré chceme dosiahnuť.

Kapitola 3

Optimalizácia

Optimalizáciu možno definovať ako proces smerujúci k nachádzaniu takých riešení, ktoré sú „lepšie“ ako súčasný stav (už dosiahnutý alebo známy). Ide o proces zlepšovania súčasného stavu, jeho výsledkom môže, ale nemusí byť optimálne riešenie [10]. Problematikou optimalizácie sa zaoberá viacero teoretických disciplín, predovšetkým operačná analýza, resp. operačný výskum. Vedecký prístup k riešeniu zložitých optimalizačných úloh predpokladá vytvorenie vhodného modelu, ktorý odráža realitu, hlavne tie jej vlastnosti, ktoré sú dôležité z hľadiska sledovaných cieľov a tiež poskytuje prostriedky hľadania optimálnych riešení.

3.1 Optimalizačný model

Optimalizačný model môžeme všeobecne zapísať funkciou

$$y = f(x_1, x_2, \cdots, x_n), \tag{3.1}$$

ktorá vyjadruje skutočnosť, že hodnota sledovanej výstupnej veličiny modelu y je funkciou kombinácie hodnôt vstupných veličín – nezávisle premenných x_i. Dôležité je, že funkciu f možno zapísať ako matematickú funkciu a hodnotu výstupnej veličiny y možno vypočítať.

Model možno zapísať aj ako vzťah medzi množinami, pri ktorom je každému prvku prvej množiny priradený určitý prvok (jeho obraz) z druhej množiny:

$$m : X \to Y \tag{3.2}$$

Prvkami množiny X sú kombinácie hodnôt vstupných veličín modelu $(x_1, x_2, \cdots, x_n)$

a prvkami množiny Y sú kombinácie hodnôt výstupných veličín $(y_1, y_2, \cdots, y_m)$. Z definície zobrazenia vyplýva, že rôznym prvkom množiny X môže byť priradený ten istý prvok množiny Y, t.j. rovnaký cieľ možno dosiahnuť rôznymi spôsobmi. Model (3.1) sa stane optimalizačným až po tom, keď doňho zahrnieme cieľ, ktorý v rámci optimalizácie sledujeme [10], napr.:

$$y = f(x_1, x_2, \cdots, x_n) \neq max \tag{3.3}$$

a podmienky, ktoré musia spĺňať vstupné veličiny x_i, zadané intervalmi, v ktorých sa hodnoty týchto jednotlivých veličín môžu pohybovať, resp. matematickými rovnicami a nerovnicami, ktorým musia vyhovovať. Uvedený vzťah sa nazýva účelová funkcia. Podmienky, ktoré musia spĺňať vstupné veličiny sa napr. v prípade úloh lineárneho programovania nazývajú vlastné odmedzenia a podmienky nezápornosti.

V prípade viackriteriálnej optimalizácie treba odpovedať na otázku, aké hodnoty musia nadobúdať jednotlivé členy vektora Y, aby riešenie optimalizačnej úlohy mohlo byť považované za optimálne, resp. aká kombinácia hodnôt y_i je optimálna, alebo „lepšia“ ako iná. Charakteristickou črtou väčšiny postupov operačnej analýzy, ktoré narábajú s modelmi uvedeného typu je, že optimálne alebo vyhovujúce riešenia sú hľadané analytickým riešením – výpočtom. (3.1)

3.2 Model simulácie

Simulácia je metóda, ktorej podstata spočíva v nahradení systému jeho simulátorom. S ním potom vykonávame pokusy. Keďže simulačný model sa používa za účelom hľadania optimálneho, alebo „čo najlepšieho riešenia“, možno ho podobne ako optimalizačný model vyjadriť funkciou

$$y = f(p_1, p_2, \cdots, p_n) \tag{3.4}$$

alebo zobrazením

$$m : P \to Y \tag{3.5}$$

Hodnoty výstupných veličín pri simulácii nemožno vypočítať, ale sú získané na výstupe simulačného modelu. V určitých prípadoch môžu byť získané ako výsledok následného

výpočtu, do ktorého môžu vstupovať okrem aspoň jednej takejto výstupnej veličiny zo simulačného experimentu aj hodnoty vstupných parametrov modelu alebo rôzne konštanty. Vstupné parametre majú často charakter náhodných veličín.

3.3 Optimalizácia založená na využití simulácie

Klasický simulačný model (3.4) sa využíva pri hľadaní optimálnych alebo „čo najlepších“ riešení ako prostriedok overovania rôznych variantov v postupnosti krokov, ktoré smerujú k optimu. Tento simulačný model nie je v skutočnosti optimalizačným, pretože neobsahuje cieľ optimalizácie. Optimalizačným sa stane po doplnení o funkciu g:

$$z = g(p_1, p_2, \cdots, p_n, \cdots, y_1, y_2, \cdots y_m), \tag{3.6}$$

ktorá sa podobne ako v prípade klasického optimalizačného modelu stane účelovou funkciou a po doplnení o cieľ optimalizácie a podmienky, ktoré musia spĺňať vstupné parametre p_i, ale niekedy aj výstupné veličiny y_j. Táto účelová funkcia môže mať napr. tvar:

$$z = g(p_1, p_2, \cdots p_n, \cdots, y_1, y_2, \cdots, y_m) \neq max \tag{3.7}$$

Cieľom môže byť aj minimalizácia funkcie, alebo čo najväčšie priblíženie sa k určitej hodnote z_{opt}. Cieľom môže byť tiež aj hľadanie takej kombinácie hodnôt parametrov modelu, ktorá vedie pri väčšom počte experimentov k čo najväčšej pravdepodobnosti, že hodnota z bude (nebude) menšia (väčšia) ako určitá zadaná hodnota z_{lim} [10].

V rovnici účelovej funkcie sa nemusia nachádzať všetky p_i a y_j, ale aby sme mohli model považovať za optimalizačný simulačný model, musí byť prítomná aspoň jedna veličina y_j. Výstupné veličiny optimalizačného simulačného modelu sú veličinami, ktorých hodnota je vyčíslená po ukončení simulačného behu. Veličiny, ktorých hodnota sa počas simulačného behu mení (napr. dĺžka radu) môže byť v účelovej funkcii prítomná len po vyhodnotení (napr. ako maximálna alebo priemerná dĺžka radu počas simulačného behu).

3.4 Optimalizačné kritériá

Pri plánovaní úloh na zdroje sa pre rozhodovanie používajú optimalizačné kritériá, ktoré umožňujú stanoviť kvalitu dosiahnutého riešenia a porovnať ho s ostatnými generovanými rozvrhmi. Tieto kritériá môžeme rozdeliť do skupín podľa parametrov, ktoré chceme optimalizovať.

- maximálny čas dokončenia úlohy $C_{max} = max(C_j, \cdots C_n)$ je čas dokončenia poslednej úlohy
- súčet časov úloh – minimalizácia súčtu časov dokončenia úloh smeruje k lepšiemu využívaniu zdrojov, skráteniu času dokončenia poslednej úlohy a ku skráteniu doby, ktorú úlohy strávia čakaním v radoch
- $\sum_{j=1}^{n} C_j$ súčet časov dokončenia úloh predstavuje minimalizáciu súčtu časov dokončenia všetkých úloh
- $\sum_{j=1}^{n} w_j C_j$ vážený súčet časov dokončenia úloh zohľadňuje váhy úloh pri minimalizácii súčtu časov dokončenia úloh.

Najväčším problémom pri plánovaní je, že hľadanie optimálneho riešenia je NP-úplný problém, ktorý sa v rozumnom čase nedá vyriešiť ak nejde o triviálne úlohy. Preto sa snažíme o nájdenie riešení približujúcich sa k optimálnemu riešeniu, ktoré sú „čo najlepšie“ a v akceptovateľom výpočtovom čase. Vo všeobecnosti sa držíme zásady [9] nesnažiť sa prehľadávať tie oblasti možných riešení, kde sa nenachádza dobrý výsledok. Ukazovateľmi pre to, kde sa má a nemá hľadať riešenie sú prirodzené obmedzenia dané typom riešenej úlohy. Ako vhodné sa ukázalo dôkladné prehľadávanie okolia nájdených dobrých riešení, v ktorom je šanca na nájdenie lepších riešení vysoká.

3.5 Klasifikácia optimalizačných metód

Optimalizačné metódy sa podľa [8] členia na tradičné (exaktné) a heuristické (približné) (obr. č. 3.1). Heuristické algoritmy sa ďalej členia na deterministické a stochastické. Stochastické na rozdiel od algoritmov deterministických využívajú v niektorých krokoch náhodné operácie, čo spôsobuje, že výsledky riešení získané

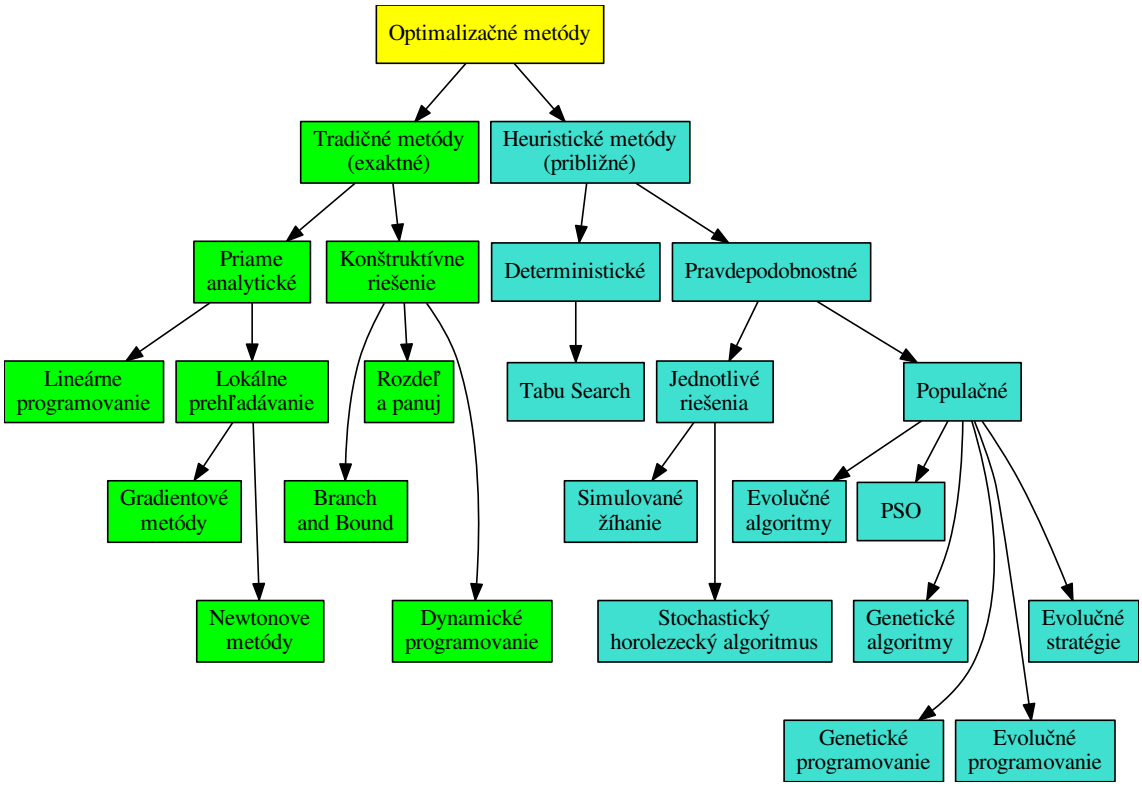
pri jednotlivých spusteniach programu sa prakticky vždy líšia. Opakované spustenie programu takto umožňuje zo získaných riešení vybrať najlepšie. Stochastické heuristické metódy poskytujú len všeobecný rámec a preto sa označujú aj ako metaheuristiky. Konkrétne operácie algoritmu sa musia zvoliť v závislosti od skúmaného problému.

Inšpiráciou pre vznik týchto metód boli prírodné procesy, preto sa nazývajú evolučnými algoritmami.

Môžeme ich rozdeliť do dvoch tried:

- metódy založené na bodovej stratégii, ich základom je operácia susedstva aktuálneho riešenia, ku ktorému sa hľadá lepšie riešenie (napr. horolezecký algoritmus, zakázané prehľadávanie),
- metódy založené na stratégii populácie (genetické algoritmy).

Na rozdiel od klasických gradientových metód pripúšťajú tieto metódy s istou pravdepodobnosťou prijatie horšieho riešenia do ďalšej iterácie. Cieľom je snaha vyhnúť sa uviaznutiu v lokálnom optime.



Obr. 3.1: Klasifikácia optimalizačných metód

3.6 Optimalizačné algoritmy

Gridové prostredie sa v ľubovoľnom čase môže meniť. Preto sa tradičné algoritmy plánovania úloh, napr. „prvý príde – prvý obslúžený“, „prvý príde – posledný obslúžený“ atď. nemusia dobre adaptovať na dynamické gridové prostredie. Existujú nové plánovacie algoritmy zamerané na skracovanie času dokončenia úloh v gridovom prostredí, ktorými sú výpočtovo náročné úlohy a dátovo náročné úlohy ošetrené rôzne a to v závislosti od situácie v gride v reálnom čase. Algoritmus priradí novú úlohu na prostriedok (zdroj) v závislosti na plánovaní úloh v minulosti a vyberie najvhodnejší prostriedok pre aktuálnu úlohu. Na zisťovanie najnovšieho stavu prostriedkov v gride sa využíva lokálna a globálna aktualizácia. Podľa aktuálnych výsledkov lokálnej a globálnej aktualizácie je možné vhodnejšie a dynamickejšie plánovanie úloh. V [2] bola navrhnutá nová štruktúra pre hodnotenie aktuálnej situácie a nový algoritmus plánovania úloh „Plánovanie úloh s adaptívnym hodnotením“ (ASJS). V experimente bol ASJS porovnaný s optimalizáciou pomocou „kolónie mravcov“, „najvhodnejšou úlohou“ a „náhodným výberom“. ASJS dosahoval kratší čas dokončenia úloh ako všetky ostatné spomenuté algoritmy. Plánovacie algoritmy pre gridové počítanie nemusia byť aplikovateľné priamo, ale môžu byť veľmi užitočné aj pri zálohovaní alebo distribúcii záťaže medzi dátovými centrami pri cloudovom počítaní, ktoré sa tiež v posledných rokoch stalo dôležitou súčasťou počítačových systémov. Jeho prístup centralizovaného dátového centra je odlišný od gridového počítania a je založené na spracovávaní pomocou transakcií na rozdiel od dávkového spracovávania v gridoch. V minulosti bolo navrhnutých veľa plánovacích algoritmov, po modifikácii je možné ich použitie aj v gridovom prostredí. Väčšinou ide o plánovacie algoritmy na princípe dávkových a on-line heuristík.

3.6.1 Dávkové heuristické plánovacie algoritmy

Úlohy sú do množiny úloh zaradené po príchode v dávkovom móde. Ich vykonanie bude naplánované plánovacím algoritmom, sú vhodnejšie pre gridové prostredie s rovnakými typmi prostriedkov. Najjednoduchším algoritmom pre plánovanie úloh je „prvý príde – prvý obslúžený“. Pri ňom sú úlohy vykonávané v poradí v akom

prišli. Druhá úloha bude vykonaná po dokončení prvej úlohy a keď dlhá úloha bude v poradí pred viacerými krátkymi úlohami, všetky krátke úlohy budú musieť čakať, kým sa dlhá úloha nedokončí. Algoritmus „najrýchlejší procesor k najväčšej úlohe“ je vhodný pre aplikácie typu „vrece úloh“. Stratégiou tohto algoritmu je plánovanie úloh podľa záťaže úlohami a výpočtového výkonu prostriedkov. V plánovacom algoritme „min-min“ je úloha vždy priradená prostriedku, ktorý ju môže dokončiť v najkratšom čase tak, aby sa dosiahlo skrátenie času na dokončenie všetkých úloh. Algoritmus „max-min“ je podobný algoritmu „min-min“ s tým, že najvyššiu prioritu dostane úloha s maximálnym najkratším časom na dokončenie. Spravodlivosť je kľúčovou myšlienkou plánovacieho algoritmu „Round-robin“. Plánovací algoritmus „rad úloh“, „rad úloh s replikáciou“ sú plánovacie algoritmy pracujúce bez poznatkov. Sú založené na stratégii plánovania úloh bez informácií, úlohy sú vyberané v náhodnom poradí a posielajú sa prostriedkom na spracovanie ihneď ako sa stanú dostupnými. Algoritmus „rad úloh s replikáciou“ zvyšuje pravdepodobnosť použitia prostriedkov s vysokým výkonom pridaním replikácie do algoritmu „rad úloh“. V algoritme „pracovný rad s replikáciou“ je každá úloha replikovaná definovaným počtom replík prenesená na dostupné prostriedky. Ak jeden prostriedok dokončí úlohu, všetky ostatné repliky, ktoré sú vykonávané inými prostriedkami budú zrušené. Výkon tohto algoritmu nie je zlý, využíva nadbytočný výpočtový výkon na vykonávanie replík, ale ak sú úlohy veľké, spotrebuje veľa času na prenos replík.

3.6.2 On-line heuristické plánovacie algoritmy

Úlohy sú naplánované keď prídu. Gridové prostredie je heterogénne, t.j. obsahuje rôzne typy prostriedkov, pre toto prostredie sú vhodnejšie „on-line heuristické plánovacie algoritmy“. Dynamický algoritmus „najrýchlejší procesor k najväčšej úlohe“ je založený na algoritme „najrýchlejší procesor k najväčšej úlohe“ a je oproti statickému upravený tak, aby bol adaptívnejší pre gridové prostredie. Algoritmus „najlepšie odpovedajúca úloha“ sa pokúša priradiť najvhodnejší prostriedok úlohe podľa hodnoty nazývanej „vhodnosť“. Rozsah vhodnosti sa nachádza v intervale od 100000 do nuly. Ďalšia dynamická plánovacia situácia v gridoch sa nazýva „plánovanie toku úloh“. Tok úloh je množina úloh na vykonávanie.Tieto úlohy môžu obsahovať dáta, ktoré musia

byť prenesené od jednej úlohy k druhej. Preto je plánovanie toku úloh zložitejšie ako jednoduché plánovanie pre množiny nezávislých úloh.

3.6.3 ACO algoritmy plánovania úloh

V posledných rokoch sa na riešenie plánovacích problémov v gridoch používa optimalizácia pomocou „kolónie mravcov“. Ide o jednoduchú gridovú simulačnú architektúru, modifikáciu základného algoritmu pre plánovanie v gride. Ide o plánovací algoritmus, ktorý vyžaduje pre plánovanie úloh informácie ako počet procesorov, počet inštrukcií za sekundu každého procesora. Tieto informácie prostriedok (zdroj) musí oznámiť monitoru prostriedkov.

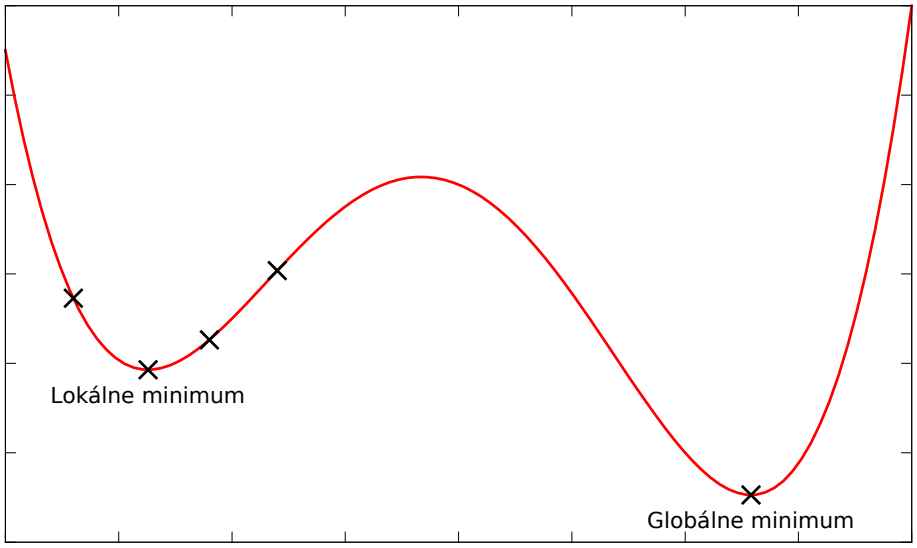
3.6.4 Plánovací algoritmus s adaptívnym hodnotením (ASJS)

„ASJS“ má za úlohu skrátiť čas na dokončenie úlohy, hodnotí nielen výkon každého prostriedku v gride, ale aj vysielací výkon každého klastra v gridovom systéme. Výpočtový výkon každého prostriedku je definovaný ako súčin rýchlosti procesorov a dostupného procesorového času a vysielací výkon je definovaný ako priemerná rýchlosť komunikácie medzi rôznymi klastrami. „ASJS“ využíva stav všetkých prostriedkov v gride ako parametre na inicializáciu hodnotenia klastrov. Hodnotenie klastra sa adaptuje použitím informácii z lokálnych a globálnych aktualizácií. Systém odošle úlohu na najvhodnejší prostriedok podľa týchto hodnotení. [4]

3.6.5 Horolezecký algoritmus

Horolezecké algoritmy (angl. „hill climbing“) sú gradientovou metódou iteračného hľadania extrému v lokálnom okolí nejakého bodu. Jeho princípom je stochastické prehľadanie lokálneho okolia najlepšieho nájdeného riešenia alebo náhodne vygenerovaného riešenia na začiatku procesu optimalizácie. Nové riešenie je prijaté, ak je lepšie než to, z ktorého vzniklo. Ak je horšie, vykoná sa iná malá zmena najlepšieho riešenia. Nové riešenie má silnú závislosť na predchádzajúcom, je to síce rýchly algoritmus, ale často uviazne v lokálnom extréme (obr. č. 3.2). Preto sa často tieto algoritmy používajú len na na dotiahnutie do extrému riešenia nájdeného inou metódou. Uviaznutiu v lokálnom extréme sa snaží vyhnúť algoritmus simulovaného

žíhania, ktorý za určitých podmienok prijíma za lepšie riešenie aj to, ktoré má mierne horšiu optimalizačnú funkciu.



Obr. 3.2: Uviaznutie v lokálnom minime

3.6.6 Tabu prehľadávanie

Algoritmus tabu prehľadávanie (angl. „tabu search“) sa snaží zamedziť vzniku krátkych cyklov pri generovaní nových rozvrhov. Jeho hlavnou myšlienkou je na rozdiel od horolezeckého algoritmu to, že keď lokálne prehľadávanie uviazne v lokálnom optime, algoritmus povolí nezlepšujúci krok. Zacykleniu sa zabraňuje pomocou konečnej pamäte – tabu zoznamu, v ktorej sa uchováva história niekoľkých posledných iterácií, ktorých opakovanie je do určitej doby zakázané. Implementovaná býva krátkodobá, prípadne aj dlhodobá pamäť a zavádza sa aj tzv. ašpiračné kritérium (podmienka ignorovania krátkodobej pamäte). Tým sa zabráni zacykleniu algoritmu, pretože ašpiračné kritérium umožňuje za určitých podmienok vykonať krok, ktorý krátkodobá pamäť zakazuje. Efektívnosť tohto algoritmu je závislá len na spôsobe definovania a spracovávania tabu zoznamu, pričom neexistujú žiadne obmedzenia v tomto smere a preto sa na rozdiel od horolezeckého algoritmu tabu prehľadávanie stáva veľmi všeobecnou stratégiou riešenia optimalizačných problémov. [9] Algoritmus tabu prehľadávania:

1. Inicializácia
   - $k = 0$.
   - náhodný výber iniciálneho rozvrhu S_0.
   - zaznamenanie doteraz nejlepšieho rozvrhu: $S_{best} = S_0$ a $best\_cost = F(S_0)$.
2. Výber a aktualizácia
   - výber kandidáta $S_c \in N(S_k)$.
   - ak je zmena $S_k \rightarrow S_c$ zakázaná, pretože je v tabu zozname potom choď na krok 2.
   - ak zmena $S_k \rightarrow S_c$ nie je zakázaná tabu zoznamom, alebo ašpiračné kritérium povolilo zmenu potom $S_{k+1} = S_c$ ulož reverznú zmenu na vrchol tabu zoznamu, posuň ďalšie pozície v tabu zozname o pozíciu ďalej, zmaž poslednú položku z tabu zoznamu, ak došlo k preplneniu zoznamu.
   - ak $F(S_c) < best\_cost$ potom $S_{best} = S_c, best\_cost = F(S_c)$.
3. Ukončenie

3.6.7 Simulované žíhanie

Patrí medzi stochastické optimalizačné algoritmy. Už z názvu tejto metódy vyplýva, že má základy vo fyzike na rozdiel od iných stochastických algoritmov, ktoré majú základ väčšinou v biológii (evolučné algoritmy). Algoritmus simulovaného žíhania je založený na analógii medzi žíhaním tuhých telies a optimalizačným problémom. Ochladzovanie pri tejto metóde znamená postupné znižovanie pravdepodobnosti, s ktorou bude do ďalšieho kroku prijaté aj horšie riešenie. Simulované žíhanie je variantom horolezeckého algoritmu, v ktorom heuristické kroky smerujúce k horšiemu riešeniu sú riadené určitou pravdepodobnosťou. Na začiatku optimalizačného procesu môžeme tento algoritmus prirovnať k stochastickému prehľadávaniu stavového priestoru, ale ku koncu priebehu je to skôr verzia horolezeckého algoritmu. Použitie tejto metódy vyplynulo zo snahy eliminovať nebezpečenstvo uviaznutia v lokálnom minime tým, že je v algoritme zavedená pravdepodobnosť náhodného skoku. Najprv

sa vykonávajú veľké náhodné zmeny a vďaka tomu sa vieme dostať z lokálneho minima. Veľkosť zmeny závisí od parametra – teploty, ktorá postupne klesá. Čím je teplota vyššia, tým sa vykonávajú väčšie zmeny. Proces simulovaného žíhania postupne pokračuje náhodným výberom susedov. Ak má riešenie nižšiu hodnotu funkcie, tak je automaticky vybrané a ďalej sa pokračuje z neho. Sused s vyššou hodnotou však tiež má šancu byť vybratý, pričom pravdepodobnosť výberu je určená funkciou:

$$e^{(-\delta/t)}, \delta \geq 0, t > 0 \tag{3.8}$$

V tejto funkcii δ vyjadruje kvalitu riešenia a t súčasnú teplotu. Algoritmus postupne znižuje túto teplotu lineárne z počiatočnej teploty na nulovú (angl. „cooling schedule“). Rozvrh chladnutia je definovaný parametrami algoritmu. Celý proces je založený na voľbe rozvrhu chladnutia a pre konkrétny problém môže byť tento teplotný rozvrh určený na začiatku behu programu tak, aby sa predišlo predčasnému schladnutiu pomocou funkcie chladnutia. Je možné vygenerovať veľký počet rozvrhov. Počet vygenerovaných rozvrhov sa rovná nastavenému počtu iterácií. Simulované žíhanie je rozšírením metódy hill climbing s cieľom vyviaznuť z lokálnych optím. Metóda vznikla v prvej polovici osemdesiatych rokov a bola inšpirovaná procesom eliminácie defektov kryštálovej mriežky kryštálov ich ohriatím s následným pomalým ochladzovaním. Na základe preddefinovanej počiatočnej teploty, ktorá sa postupne znižuje podľa plánu ochladzovania, majú aj slabšie susedné riešenia šancu byť vybraté. To znamená, že je určitá pravdepodobnosť, že smer postupu od jedného stavu k nasledujúcemu už nemusí byť len jedným smerom (od horšieho k lepšiemu), ale s určitou pravdepodobnosťou je možný aj ľubovoľný iný prechod. [5]

Nevýhodou simulovaného žíhania je, že treba nastavovať veľa vstupných parametrov a od ich správnej voľby závisí kvalita získaných výsledkov. Napríklad vysoká hodnota parametra j_{max} (počet prehľadávaní pri danej teplote t_k) spomaľuje algoritmus, zatiaľ čo nízka hodnota spôsobí, že prehľadávanie pravdepodobne skončí v nejakom lokálnom minime. Ďalšími parametrami sú počiatočná teplota T_{max} a koncová teplota T_{min}. T_{max} je najlepšie zvoliť tak, aby počet akceptovaných zhoršujúcich stavov bol približne polovičný. Za predpokladu vysokej teploty by algoritmus prijal takmer všetky zhoršujúce rozvrhy, podobne ako pri hľadaní globálneho optima pomocou slepého

prehľadávania. Pri malých hodnotách T_{max} sa zas dosahujú výsledky podobné ako pri horolezeckom algoritme. Algoritmus simulovaného žíhania [9]:

1. Inicializácia
   - $k = 0$.
   - náhodný výber počiatočného rozvrhu S_0.
   - zaznamenanie doteraz nejlepšieho rozvrhu: $S_{best} = S_0$ a $best\_cost = F(S_0)$.
   - nastavenie počiatočnej teploty $t_0 = T_{max} > 0$.
   - výber funkcie chladnutia (t).
2. Výber a aktualizácia
   - výber kandidáta $S_c \in N(S_k)$.
   - ak $F(S_c) < best\_cost$ potom $S_{k+1} = S_c, S_{best} = S_c, best\_cost = F(S_c)$ a skok na krok 3.
   - ak $F(S_c) \geq best\_cost$ potom
     - ak $F(S_c) < F(S_k)$ potom $S_{k+1} = S_c$
     - inak generuj náhodné číslo U_k
     - ak $U_k < e^{\frac{F(S_c)-F(S_k)}{t_k}}$ potom $S_{k+1} = S_c$.
     - inak $S_{k+1} = S_k$.
3. Aktualizácia teploty t_k
   - $t_k = \alpha(t_k)$.
4. Ukončenie
   - ak platia podmienky ukončenia potom algoritmus končí ($t_k \leq T_{min}$, alebo skôr spomenuté kritériá).
   - inak $k = k + 1$ a skok na krok 2.

Okrem klasického simulovaného žíhania existuje aj tzv. rýchle simulované žíhanie (angl. „fast simulated annealing“ – FSA). Rozdiel spočíva v rozdelení pravdepodobnosti náhodných zmien. Kým SA používa Gaussovo rozdelenie, FSA používa Cauchyho rozdelenie, ktoré nemá definovanú strednú hodnotu, rozptyl a pod. a tým sa zvyšuje pravdepodobnosť úspešného riešenia.

3.6.8 Genetický algoritmus

Genetický algoritmus pracuje na princípe inšpirovanom Darwinovou evolučnou teóriou. Ide tu o snahu skombinovať dobrých jedincov (jednotlivé riešenia) z populácie (množiny všetkých možných riešení) tak, aby vznikol lepší jedinec (riešenie). Podstata algoritmu spočíva v zakódovaní informácie pri hľadaní tak, aby slabšia generácia bola pri každej iterácii nahradená lepšou generáciou. Využívajú sa pri tom tri operátory:

- selekcia
- kríženie
- mutácia

Pri selekcii sa vyberajú jedinci podľa ich kvality, čím je jedinec lepší, tým je vyššia pravdepodobnosť, že bude vybratý. Pri krížení dochádza ku kombinácii vlastností (rozvrhov) dvoch alebo viacerých jedincov, čím vzniká nový jedinec. Mutácia znamená modifikáciu jedného jedinca, čím sa dá zabrániť uviaznutiu v lokálnom extréme. [11]

Kapitola 4

Návrh modelu optimalizácie plánovania úloh

4.1 Simulátor plánovania úloh

Zostaveniu samotného algoritmu plánovania úloh predchádza vytvorenie simulátora plánovania úloh. Implementovaný simulátor bol napísaný v jazyku C. Na vstup simulátora prostredníctvom optimalizačného algoritmu prichádzajú informácie o poradí úloh vykonávaných na každom prostriedku (zdroji), o závislosti úloh na súboroch, o veľkosti jednotlivých súborov atď. Vstupy sú náhodne generované generátorom vstupov, ktorý bol tiež napísaný v jazyku C.

Vygenerovaná zostava vstupov obsahuje:

- informácie o spojeniach medzi uzlami siete,
- rýchlosti procesorov,
- dĺžke úloh,
- veľkosti súborov,
- informácie o súboroch generovaných úlohami,
- o závislosti úloh na súboroch,
- počiatočné umiestnenia súborov.

Počet uzlov siete, počet úloh a počet súborov sa nastavuje manuálne pred začatím procesu generovania. Maximum počtu uzlov, úloh a súborov je dané kapacitou

operačnej pamäte. Jednoduchý príklad výstupu generátora je uvedený na obr. 4.1 Vstup. Vstup obsahuje:

- vlastnosti uzlov gridu,
- sieťové spojenia medzi nimi,
- vlastnosti úloh,
- vlastnosti súborov.

Simulátor načítané vstupy využíva pri simulácii. Pre každý uzol siete a úlohu vypočíta čas dokončenia prenosu každého súboru, čas začatia vykonávania úlohy a čas ukončenia vykonávania úlohy. Informácie získané simuláciou simulátor vráti ako svoj výstup, ktorý je zároveň vstupom pre optimalizačný algoritmus. Tieto informácie majú podobu matíc, jednoduchý príklad je uvedený na obr. 4.2 Výstup. Výstupom simulátora je:

- čas spustenia každej úlohy,
- čas dokončenia prenosu každého súboru.

Výstup simulátora bude slúžiť pre potreby vyhodnocovania času vykonávania úloh a plánovanie úloh pomocou optimalizačného algoritmu. Simulátor je znázornený vývojovým diagramom na obr. 4.5 Hlavná funkcia simulátora je na obr. 4.3 a verzia pre GPU je na obr. 4.4. [4]

Tvorba simulátora plánovania úloh je prvou, ale nosnou časťou modelu optimalizácie, pretože definuje cieľ, ktorý chceme optimalizáciou dosiahnuť. Optimalizačnú funkciu zvolenú ako kombináciu dvoch optimalizačných kritérií: čas dokončenia poslednej úlohy a súčet časov dokončenia všetkých úloh môžeme zapísať v tvare

$$f = t_{last} * J + \sum_{j=1}^{J} t_j, \tag{4.1}$$

kde t_{last} je čas dokončenia poslednej úlohy, J je počet úloh a t_j je čas dokončenia úlohy j.

Simulátor pracuje nasledovne:

1. inicializácia

```
node_latency
matrix 4 4
28 34 14 21
34 26 48 13
14 48 28 35
21 13 35 19
cpu_speed
matrix 4 1
6 11 13 9
job_length
matrix 8 1
262 949 67 247 428 548 711 298
file_size
matrix 12 1
4373 6697 9370 410 3906 164 4382 4089 2056 5428 7905 3865
file_genby
matrix 12 1
-1 3 4 -1 3 -1 -1 -1 -1 1 -1 -1
job_files
matrix 8 12
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
1 1 0 0 0 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 1 1 0 0 0
file_mirrors
matrix 12 4
-1 -1 -1 -1 -1 -1 -1 -1 -1 0 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 0 0 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 0 -1 -1 -1 -1 -1 -1 -1
```

Obr. 4.1: Vstup

```
matrix 4 1
44 118 100 30
matrix 12 4
-1 -1 -1 -1 -1 -1 -1 -1 -1 203 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 111 -1 -1
-1 119 152 -1 119 -1 -1 -1 -1 94 -1 -1
-1 -1 -1 -1 0 -1 -1 -1 -1 136 -1 -1
```

Obr. 4.2: Výstup

```
int opt_sim()
{
  int n, j, f, dn, fi, maxtime = 0, ji, this_file, all_files,
     jobstart, jobwait, jobfinish, improving, node_not_finished =
     1, job_done = 1, job_improved, file_download, c_lastjobfinish,
     prev_last;
  mtx_set(M(NODECOMPLETEDJOBS), 0);
  mtx_set(M(NODELASTJOBFINISH), 0);
  mtx_set(M(NODEFILEDOWNLOAD), -1);
  mtx_set(M(NODEEDOWNLOADCOUNT), 0);
  mtx_set(M(NODEEDOWNLOADTIMES), -1);
  mtx_set(M(NODEFILEDOWNLOADFOR), -1);
  while (node_not_finished || job_improved) {
    if (!job_done && node_not_finished) return -1;
    node_not_finished = 0;
    job_done = 0; job_improved = 0;
    for (n=0; n<nodecount; n++) {
      prev_last = E(NODELASTJOBFINISH, n, 0);
      c_lastjobfinish = 0; file_download = 0;
      for (ji=0; ji<MTX(M(NODEJOBCOUNT),n,0); ji++) {
        improving = (ji < E(NODECOMPLETEDJOBS, n, 0));
        j = E(NODEJOBS, n, ji); all_files = 0;
        for (fi=0; fi<E(JOBFILESCOUNT, j, 0); fi++) {
          f = E(JOBFILESLIST, j, fi);
          if ((this_file = E(FILE_MIRRORS, f, n)) >= 0) {
            if (E(NODEFILEDOWNLOADFOR, n, f) != j) continue;
            dn = job_selecttransfernode(n, f, file_download - 1);
            if (dn < 0) { node_not_finished = 1; goto nextnode; }
            this_file = job_filetransferfinishtime(dn, n, f,
               file_download - 1);
            if (E(NODEEDOWNLOADTIMES, n, file_download) >
               this_file) job_improved = 1; E(NODEEDOWNLOADTIMES,
               n, file_download) = this_file; E(FILE_MIRRORS, f, n)
               = this_file; E(NODEFILEDOWNLOADFOR, n, f) = j;
          if (this_file > all_files) all_files = this_file;
             file_download++; }
        jobstart = c_lastjobfinish;
        jobwait = all_files > jobstart ? all_files - jobstart : 0;
        if (all_files > jobstart) jobstart = all_files;
        jobfinish = jobstart + job_lengthonnode(n, j);
           c_lastjobfinish = jobfinish;
        MTX(M(NODELASTJOBFINISH), n, 0) = c_lastjobfinish;
        MTX(M(JOBFINISH), j, 0) = jobfinish;
        if (!improving) { MTX(M(NODECOMPLETEDJOBS), n, 0) ++;
           job_done++; }
        for (f=0; f<filecount; f++) {
          if (E(FILE_GENBY, f, 0) == j) {
            E(FILE_MIRRORS, f, n) = jobfinish;
            E(NODEFILEDOWNLOADFOR, n, f) = j;
          } } } nextnode: {} } }
  for (n=0; n<nodecount; n++) { if (E(NODELASTJOBFINISH, n, 0) >
     maxtime) maxtime = E(NODELASTJOBFINISH, n, 0); }
  return maxtime;
}
```

Obr. 4.3: Hlavná funkcia simulátora

```
static __global__ void opt_sim_k(
  int nodecount, int jobcount, int filecount, int
     *nodelastjobfinish, int *nodejobcount, int *nodecompletedjobs,
     int *nodejobs, int *jobfilescount, int *jobfileslist, int
     *filemirrors, int *nodefiledownloadfor, int
     *nodeedownloadtimes, int *nodelatency, int *nodebandwidth, int
     *filesize, int *joblength, int *cpuspeed, int *filegenby, int
     *jobfinishtime, int *kernelstatus
  )
{
  int ji, fi, j, f, this_file, dn, job_improved = 0, jobstart,
     jobfinish, job_done = 0, node = blockIdx.x * blockDim.x +
     threadIdx.x; if (node >= nodecount) return; int
     c_lastjobfinish = 0; int file_download = 0;
  for (ji = 0; ji<E(nodejobcount, nodecount, node, 0); ji++) {
    int improving = (ji < E(nodecompletedjobs, nodecount, node, 0));
    j = E(nodejobs, nodecount, node, ji); int all_files = 0;
    for (fi = 0; fi < E(jobfilescount, jobcount, j, 0); fi++) {
      f = E(jobfileslist, jobcount, j, fi);
      if ((this_file = E(filemirrors, filecount, f, node)) >= 0) {
        if (E(nodefiledownloadfor, nodecount, node, f) != j) {
           continue; } }
      if (node==0 && ji==2 && fi==1) kernelstatus[KS_DEBUG] = -2;
      dn = job_selecttransfernode_g(nodecount, jobcount, filecount,
         filemirrors, nodeedownloadtimes, nodelatency, filesize,
         nodebandwidth, node, f, file_download - 1);
      if (dn < 0) { kernelstatus[KS_NODE_NOT_FINISHED] = 1; goto
         end; }
      this_file = job_filetransferfinishtime_g(nodecount, jobcount,
         filecount, filemirrors, nodeedownloadtimes, nodelatency,
         filesize, nodebandwidth, dn, node, f, file_download - 1);
      int file_oldtime = E(nodeedownloadtimes, nodecount, node,
         file_download);
      if (file_oldtime > this_file) { job_improved = 1; }
      if (file_oldtime != this_file) { E(nodeedownloadtimes,
         nodecount, node, file_download) = this_file;
         E(filemirrors, filecount, f, node) = this_file; }
      if (!improving) E(nodefiledownloadfor, nodecount, node, f) =
         j; if (this_file > all_files) all_files = this_file;
         file_download++; } jobstart = c_lastjobfinish; if
         (all_files > jobstart) jobstart = all_files; jobfinish =
         jobstart + job_lengthonnode_g(nodecount, jobcount,
         filecount, joblength, cpuspeed, node, j);
    c_lastjobfinish = jobfinish;
    E(nodelastjobfinish, nodecount, node, 0) = c_lastjobfinish;
    E(jobfinishtime, jobcount, j, 0) = jobfinish;
    if (!improving) { E(nodecompletedjobs, nodecount, node, 0) ++;
       job_done = 1; }
    for (f=0; f<filecount; f++) { if (E(filegenby, filecount, f, 0)
       == j) { E(filemirrors, filecount, f, node) = jobfinish; } }
       } end:
  if (job_done) kernelstatus[KS_JOB_DONE] = 1;
  if (job_improved) kernelstatus[KS_JOB_IMPROVED] = 1;
}
```

Obr. 4.4: Hlavná funkcia simulátora - GPU

2. prejdi každým uzlom
    - prejdi každou úlohou
        - prejdi každým súborom, na ktorom úloha závisí
            * ak je súbor už prítomný na uzle pokračuj ďalším súborom
            * nájdi uzol, z ktorého možno najskôr stiahnuť súbor
            * zaznamenaj presun súboru
            * ak nebolo možné preniesť niektorý súbor pokračuj ďalším uzlom
        - vypočítaj čas dokončenia úlohy
        - zaznamenaj dokončenie úlohy a vytvorenie súborov
3. ak niektorá úloha nebola dokončená alebo došlo k zlepšeniu vráť sa na krok 2

4.2 Model optimalizácie plánovania úloh

4.2.1 Opis návrhu

Model optimalizácie plánovania úloh sa skladá z modelu prostredia, špecifikácie úloh a rozvrhu. Model prostredia špecifikuje parametre uzlov gridu a sietí medzi nimi. Špecifikácia úloh pozostáva z času potrebného na vykonanie úloh, súborov, na ktorých úlohy závisia, súboroch generovaných úlohami a veľkosti súborov. Rozvrh sa skladá z usporiadaného zoznamu úloh pre každý uzol a zoznamu určujúceho poradie sťahovania súborov pre každú úlohu. Vyhodnocovacia funkcia vráti dve hodnoty – čas dokončenia poslednej úlohy a súčet časov dokončenia každej úlohy.

4.2.2 Výber modelu optimalizácie

Existujú rôzne metódy globálnej optimalizácie. Na globálnu optimalizáciu sú vhodnejšie heuristické metódy (približné) ako tradičné metódy (exaktné). Heuristiku je možné opísať ako postup, pri ktorom sa hľadá riešenie na základe skúseností, ktorý poskytuje riešenie v prijateľnom čase, ale nezaručuje správny výsledok. Tento postup sa používa v prípade, keď na riešenie úlohy neexistuje jednoznačný algoritmus, alebo je hľadanie pomocou jednoznačného algoritmu príliš zdĺhavé. Vzhľadom k tomu sa

musíme častokrát pri globálnej optimalizácii uspokojiť s tým, že pri hľadaní budeme úspešní len s určitou pravdepodobnosťou. Preto je veľmi dôležité určiť, aký minimálny počet pokusov musíme vykonať, aby sme minimalizovali hladinu rizika neúspechu. Existuje veľa algoritmov, ktoré sa dajú využiť pri riešení optimalizačných úloh, napr. evolučné algoritmy, genetické algoritmy, slepé prehľadávanie, horolezecké algoritmy, simulované žíhanie atď.

Pri vývoji simulátora bol najprv použitý horolezecký algoritmus, ktorý často uviazol v lokálnom minime (obr.č. 4.6). Preto pre navrhnutý model optimalizačného algoritmu (obr.č.4.7, obr.č. 4.8) bola zvolená metóda simulovaného žíhania (SA), ktorá znižuje pravdepodobnosť uviaznutia v lokálnom minime. Výhoda tejto metódy spočíva aj v jej jednoduchosti a malej pamäťovej náročnosti.

4.3 Implementácia plánovacieho algoritmu

Relatívnou novinkou a ešte stále pomerne málo etablovanou myšlienkou je myšlienka využitia grafického hardvéru ako matematického koprocesora k hlavnému centrálnemu procesoru (CPU). Po prvý krát bola v tejto súvislosti použitá skratka GPGPU v roku 2002. Je to skratka z anglického General Purpose Computing on Graphics Processors – vykonávanie všeobecných výpočtov prostredníctvom grafických procesorov. CPU a GPU sú dve architektúry, ktoré sa navzájom doplňujú, pričom CPU sa stará o algoritmus a GPU o spracovanie dát. Efektívnosť GPU sa prejavuje najmä v maticovej aritmetike a predovšetkým pri paralelných dátových operáciách. V prípade, že nie je možné problém paralelizovať, je lepšie riešiť ho klasicky, ak sa dá riešiť paralelne mal by sa riešiť pomocou GPU. 4.9

Potreba efektívne využívať výkon počítačového systému vedie programátorov k vyvíjaniu efektívnejších algoritmov a ich implementácií. Princípy GPGPU prinášajú do tejto problematiky nové možnosti riešení. S výraznou podporou výrobcov grafických procesorov majú programátori možnosť nielen experimentovať, ale aj prakticky využívať výhody GPGPU pri vývoji svojich produktov. Využívanie výkonu GPU sa stalo populárnym hlavne vo vedeckých aplikáciách z dôvodu potreby spracovania obrovského množstva dát a pre možnosť paralelizácie problémov. Používanie GPGPU

má význam len vtedy, ak pracujeme s väčším objemom údajov.

Základom techniky GPGPU je model prúdového spracovania, pri ktorom sa množina vstupných údajov spracováva sériou operácií – kernelov. Základnou podmienkou je možnosť spracovania každého z prvkov samostatne, nezávisle od spracovávania ostatných prvkov vstupnej množiny. GPU používa SIMT architektúru, pri ktorej beží rovnaký kód na veľkom počte vlákien.

Existuje niektoľko programovacích prostredí pre GPGPU, napríklad CUDA, alebo OpenCL. Optimalizačný algoritmus bol implementovaný v jazyku C. Optimalizačná funkcia, ktorá tvorí podstatnú časť programu, môže byť spustená aj na grafickej karte v prostredí CUDA. CUDA od spoločnosti NVIDIA je GPU architektúra, ktorá umožňuje využiť shader procesory pre paralelné GPGPU výpočty. Je to softvérová a hardvérová architektúra podporujúca využívanie grafických kariet na ľubovoľné výpočty bez nutnosti pristupovať ku GPU cez grafické rozhranie. Podstatnou vlastnosťou tejto architektúry je rýchly prenos dát medzi CPU a GPU. [6]

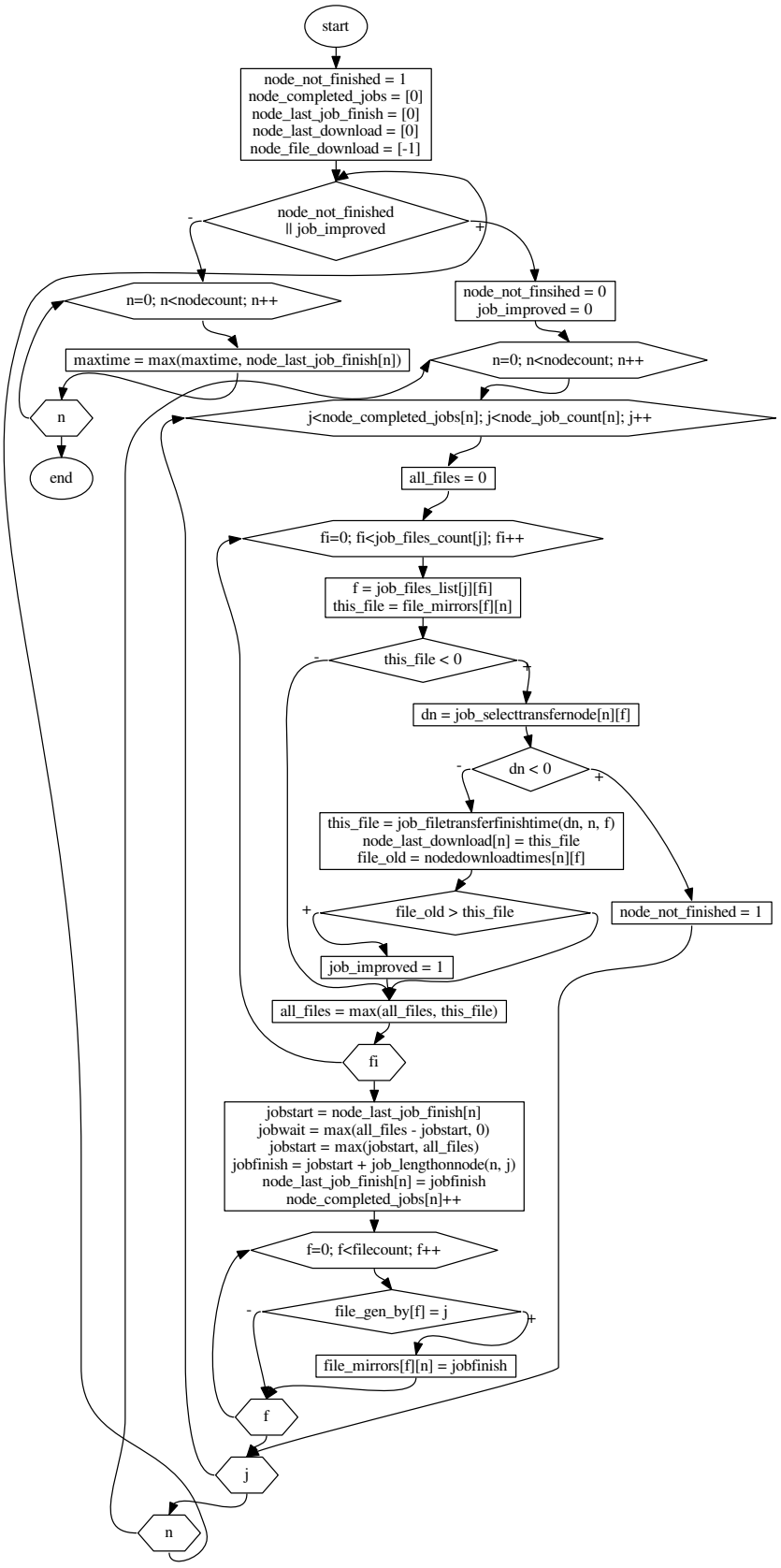
Problém bol paralelizovaný presunutím optimalizačnej funkcie na GPU. Každý uzol je vyhodnocovaný vo svojom vlastnom vlákne. Pre každý uzol prejdeme všetkými jeho úlohami a vypočítame najskorší možný čas stiahnutia každého potrebného súboru. Vypočítame čas začiatku behu úlohy ako maximum z času dokončenia predchádzajúcej úlohy a času dokončenia prenosu všetkých potrebných súborov. Potom vypočítame čas behu úlohy ako podiel časovej náročnosti úlohy a rýchlosti uzla. Toto opakujeme až kým nedôjde k úspešnému dokončeniu všetkých úloh a čas sa prestane zlepšovať.

4.4 Overenie modelu optimalizácie

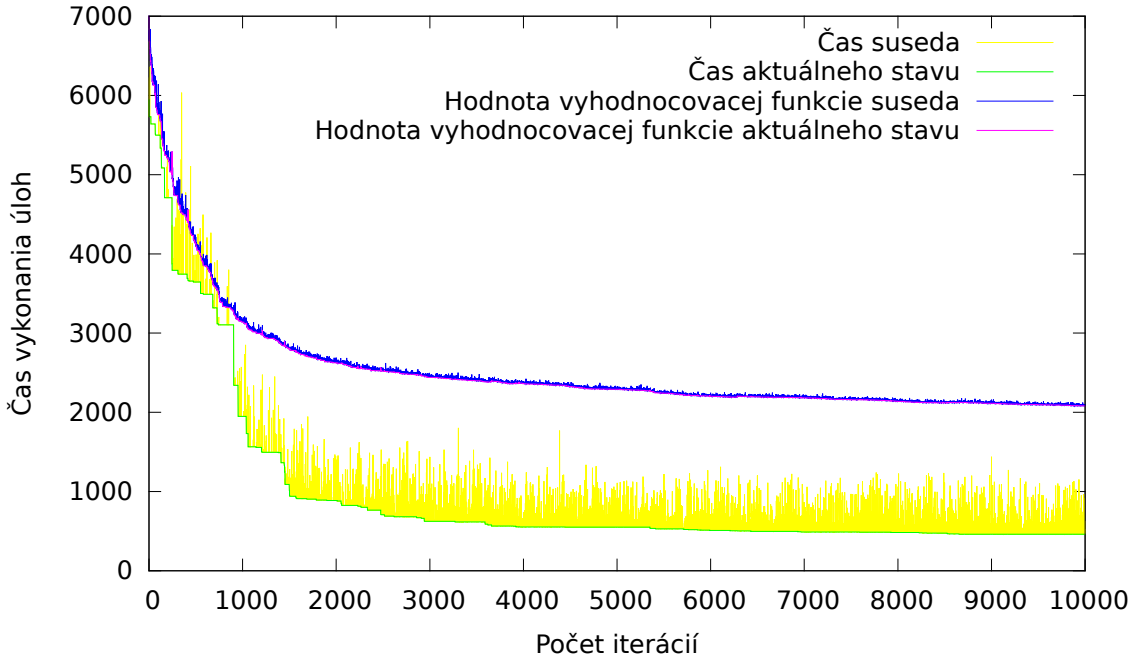
Rýchlosť implementácie programu na CPU a GPU bola porovnaná spustením na počítači s procesorom Intel i7-720QM a grafickou kartou NVIDIA GeForce GTS 360M pre počty uzlov od 32 do 704 . Výsledky meraní sú uvedené v tabuľke č. 5.1 a grafe 5.1.

Ako príklad procesu optimalizácie je uvedený priebeh optimalizácie pri 128 uzloch a počte 60000 iterácií na obr. č. 5.2. Ďalej bol overený priebeh optimalizácie v závislosti na počte iterácií na počítači s procesorom Intel i5-4570. Výsledky sú znázornené na

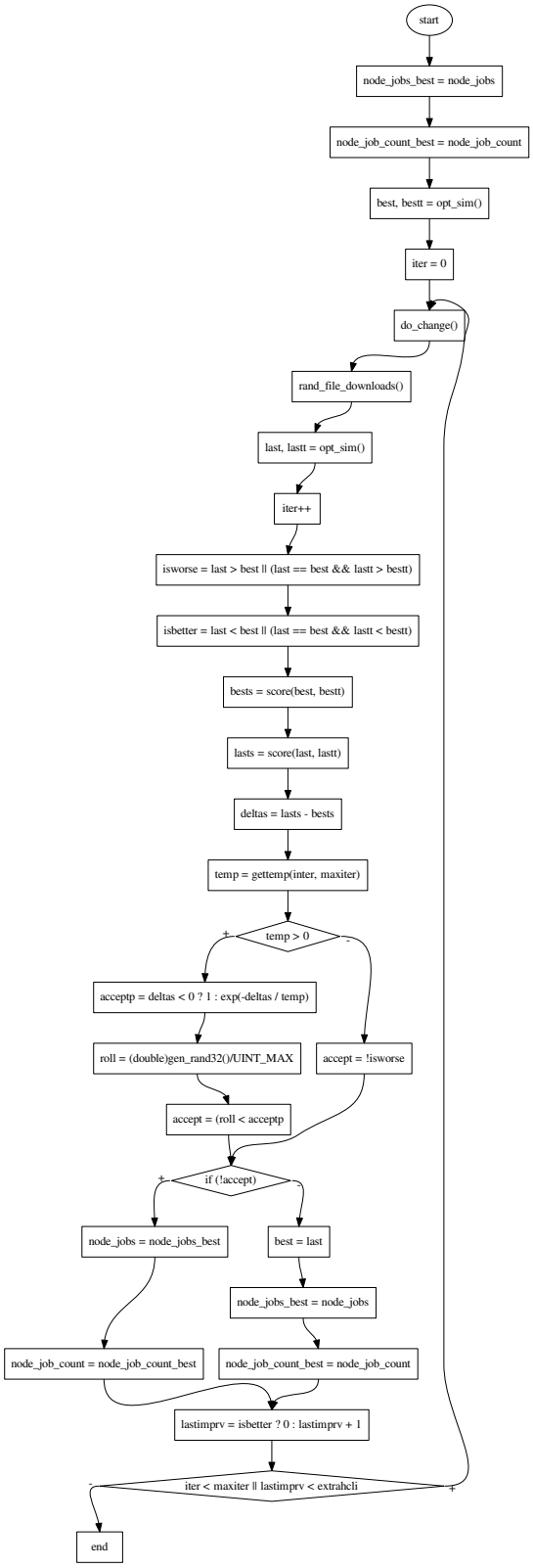
obr. 5.3, 5.4, 5.5.



Obr. 4.5: Simulátor



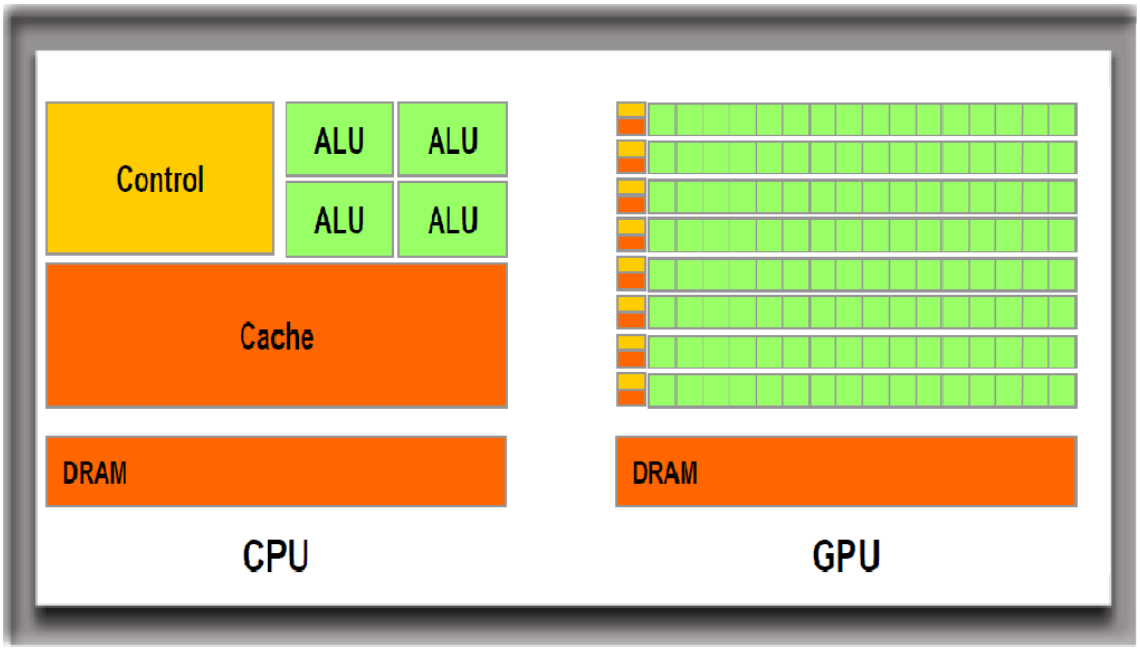
Obr. 4.6: Priebeh optimalizácie horolezeckým algoritmom



Obr. 4.7: Optimalizačný algoritmus

```
void opt()
{
  int best, last, i, iter, depok, code, accept, lastimprv = 0;
  double temp, acceptp, roll;
  uint64_t bestt, lastt, bests, lasts;
  int64_t deltas;
  assign_jobs_randomly();
  mtx_copy(M(NODEJOBSBEST), M(NODEJOBS));
  mtx_copy(M(NODEJOBCOUNTBEST), M(NODEJOBCOUNT));
  best = opt_sim_c(&bestt);
  for (iter = 0; iter < maxiter || lastimprv < extrahcli;) {
    depok = 1;
    int changes = gen_rand32() % 4 + 1;
    for (i=0; i<changes; i++) depok = depok && do_change();
    rand_file_downloads();
    if (!depok) {
      mtx_copy(M(NODEJOBS), M(NODEJOBSBEST));
      mtx_copy(M(NODEJOBCOUNT), M(NODEJOBCOUNTBEST));
      printf("d␣%d\n", best); continue; }
    last = opt_sim_c(&lastt);
    if (last == -1) {
      mtx_copy(M(NODEJOBS), M(NODEJOBSBEST));
      mtx_copy(M(NODEJOBCOUNT), M(NODEJOBCOUNTBEST));
      printf("D␣%d\n", best);
      continue; } iter++;
    int isworse = last > best || (last == best && lastt > bestt);
    int isbetter = last < best || (last == best && lastt < bestt);
    bests = score(best, bestt); lasts = score(last, lastt);
    deltas = lasts - bests; temp = gettemp(iter, maxiter);
    if (temp > 0) {
      acceptp = deltas < 0 ? 1 : exp(-deltas / temp);
      roll = (double)gen_rand32()/UINT_MAX;
      accept = (roll < acceptp);
    } else { accept = !isworse; }
    if (!accept) {
      mtx_copy(M(NODEJOBS), M(NODEJOBSBEST));
      mtx_copy(M(NODEJOBCOUNT), M(NODEJOBCOUNTBEST));
      code = '-';
    } else {
      best = last; bestt = lastt;
      mtx_copy(M(NODEJOBSBEST), M(NODEJOBS));
      mtx_copy(M(NODEJOBCOUNTBEST), M(NODEJOBCOUNT));
      code = '+'; }
    if (isbetter) { lastimprv = 0; } else { lastimprv++; }
    printf("%c␣%6d/%d␣%d␣%d␣|␣%" PRIu64 "␣%" PRIu64 "␣|␣%" PRIu64 "␣
       %" PRIu64 "␣%8" PRIi64 "␣|␣%3.0f%%␣t=%f␣A%c␣%f␣LI=%d\n",
      code, iter, maxiter, best, last, bestt, lastt, bests, lasts,
         deltas, acceptp * 100, temp, accept ? '+' : '-', roll,
         lastimprv);
    fprintf(logfile, "%d␣%d␣%" PRIu64 "␣%" PRIu64 "␣%" PRIu64 "␣%"
       PRIu64 "␣%f\n", best, last, bestt, lastt, bests, lasts,
       temp);
  } }
```

Obr. 4.8: Simulované žíhanie



Obr. 4.9: Štruktúra CPU a GPU

Kapitola 5

Experimentálne overenie

Cieľom diplomovej práce bolo overenie vhodnosti použitia pokročilých plánovacích algoritmov pre plánovanie úloh v gridovom prostredí a implementácia vybraného algoritmu na GPGPU. Ako nástroj zvoleného riešenia bol vybratý centralizovaný plánovač, nakoľko na grafickej karte je najvhodnejšie spracovanie veľkého množstva dát. Pri návrhu plánovača bolo zohľadnené, že gridové zdroje sú len výpočtové zdroje, všetky úlohy majú tiež len výpočtový charakter a zohľadňuje sa prenos súborov medzi úlohami. Neuvažovalo sa s preemptívnymi (prerušiteľnými) úlohami, to znamená, že všetky úlohy sú nepreemptívne (neprerušiteľné). Dĺžka výpočtu úloh je známa dopredu, počas rozvrhovania sa nemení. Posielanie úloh na zdroje realizuje plánovač (optimalizátor).

Počas experimentu sa testoval navrhnutý algoritmus zrýchlenia výpočtov pomocou grafickej karty v porovnaní s CPU. Vzhľadom k náročnosti výpočtu očakávanej (predpokladanej) hodnoty zrýchlenia výpočtu teoretická hodnota zrýchlenia nebola odhadnutá. Experiment prebiehal v prostredí NVIDIA CUDA.

Rýchlosť implementácie programu na CPU a GPU bola porovnaná spustením na počítači s procesorom Intel i7-720QM a grafickou kartou NVIDIA GeForce GTS 360M pre počty uzlov od 32 do 704. Počet uzlov pre expertiment bol zvolený tak, aby dĺžka behu programu bola vhodná pre účely testovania. Pri pokusoch s nižším počtom uzlov ako 32 program bežal príliš krátko na to, aby výsledok behu programu na CPU a GPU malo zmysel porovnávať. Pri vyššom počte uzlov ako 704 program bežal príliš dlho.

5.1 Metodika merania

Vzhľadom k tomu že implementovaný algoritmus pracuje s funkciou založenou na náhode a na celkové výsledky vplýva generátor náhodných čísel, bol vo vybraných prípadoch spustený opakovane. Keďže pri opakovaných spusteniach sa výsledok výrazne nelíšil neboli ďalšie testy opakované. Inicializácia generátora náhodných čísel bola vykonaná pri každom teste s iným semienkom vybraným náhodne. Experimenty boli spúšťané na náhodne vygenerovaných dátach.

5.2 Výsledky meraní

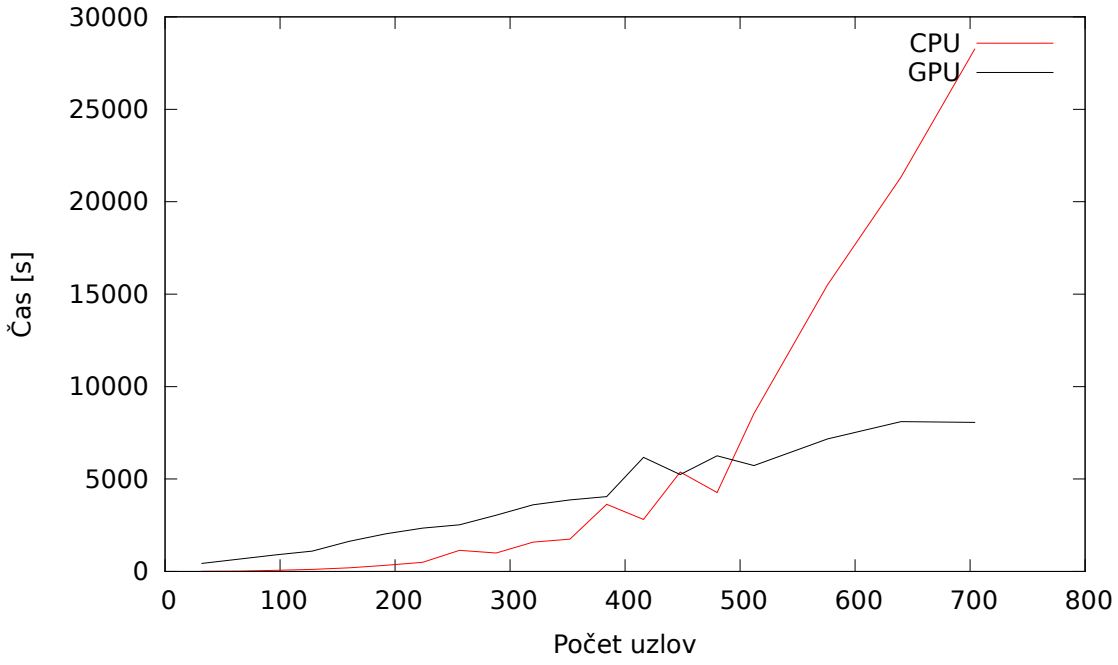
Výsledky meraní sú uvedené v tabuľke č. 5.1. V stĺpcoch je uvedený počet uzlov, úloh a súborov, čas vykonávania programu na CPU a GPU v sekundách a zrýchlenie vykonávania programu na GPU oproti CPU. V riadkoch sú jednotlivé testovacie prípady. Z tabuľky sa ukazuje, že pri 512 uzloch začína byť algoritmus na GPU výrazne rýchlejší a so zvyšujúcim sa počtom uzlov sa zrýchlenie ďalej zvyšuje, pri 704 uzloch dosiahol 350%-né zrýchlenie.

Na obr. č. 5.1 je graficky znázornený čas vykonávania programu na CPU a GPU v závislosti od počtu uzlov, na obr. č. 5.2 je znázornený priebeh optimalizácie pri 128 uzloch a počte 60000 iterácií. Na obr. 5.3, 5.4, 5.5 sú znázornené priebehy optimalizácie pre 128 uzlov a rôzne počty iterácií.

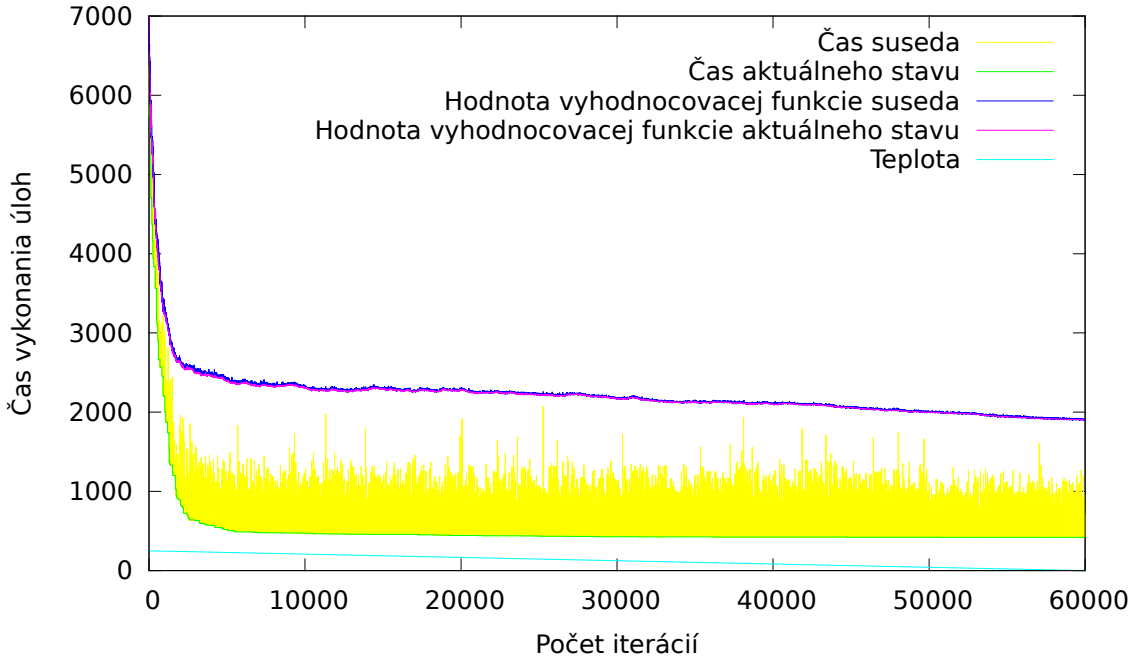
Pri experimente sa ukázalo, že je možné zrýchlenie plánovania úloh pomocou GPU, tiež sa ukázalo, že algoritmus simulovaného žíhania je možné použiť na plánovanie úloh.

Tabuľka 5.1: Porovnanie CPU a GPU

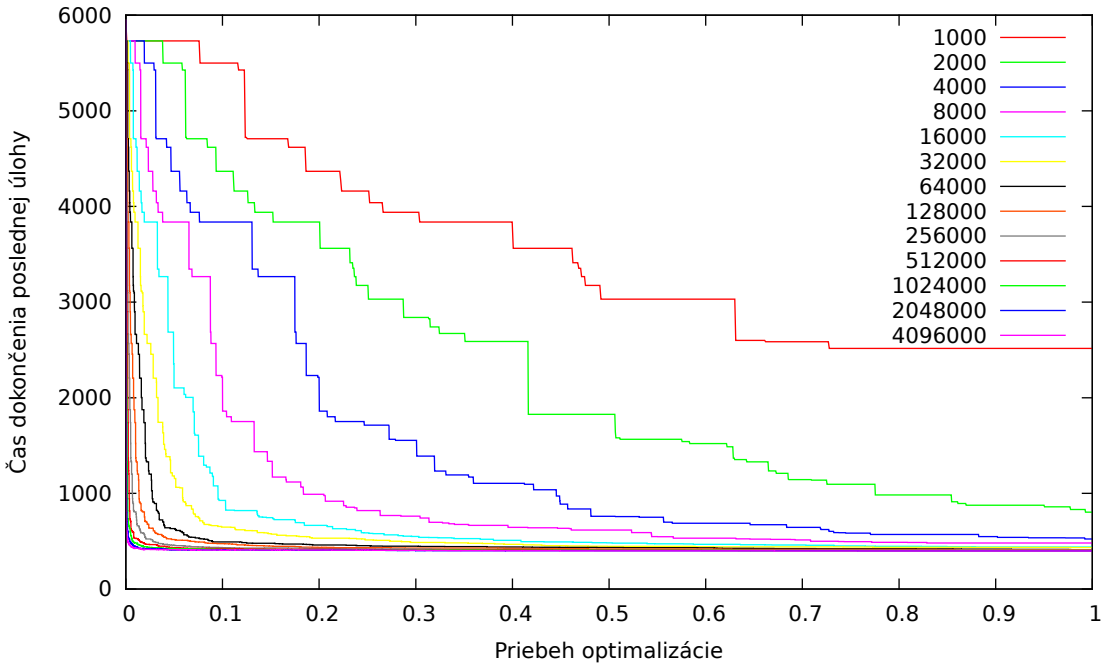
Počet			Čas [s]		
Uzlov	Úloh	Súborov	CPU	GPU	Zrýchlenie
32	256	512	7	463	0.01
64	512	1024	16	661	0.02
96	768	1536	59	895	0.07
128	1024	2048	115	1096	0.10
160	1280	2560	199	1625	0.12
192	1536	3072	330	2040	0.16
224	1792	3584	491	2340	0.21
256	2048	4096	1142	2520	0.45
288	2304	4608	998	3041	0.49
320	2560	5120	1588	3603	0.44
352	2816	5632	1751	3869	0.45
384	3072	6144	3632	4048	0.90
416	3328	6656	2809	6170	0.46
448	3584	7168	5380	5245	1.03
480	3840	7680	4270	6256	0.68
512	4096	8192	8538	5728	1.49
576	4608	9216	15500	7167	2.16
640	5120	10240	21337	8108	2.63
704	5632	11264	28257	8062	3.50



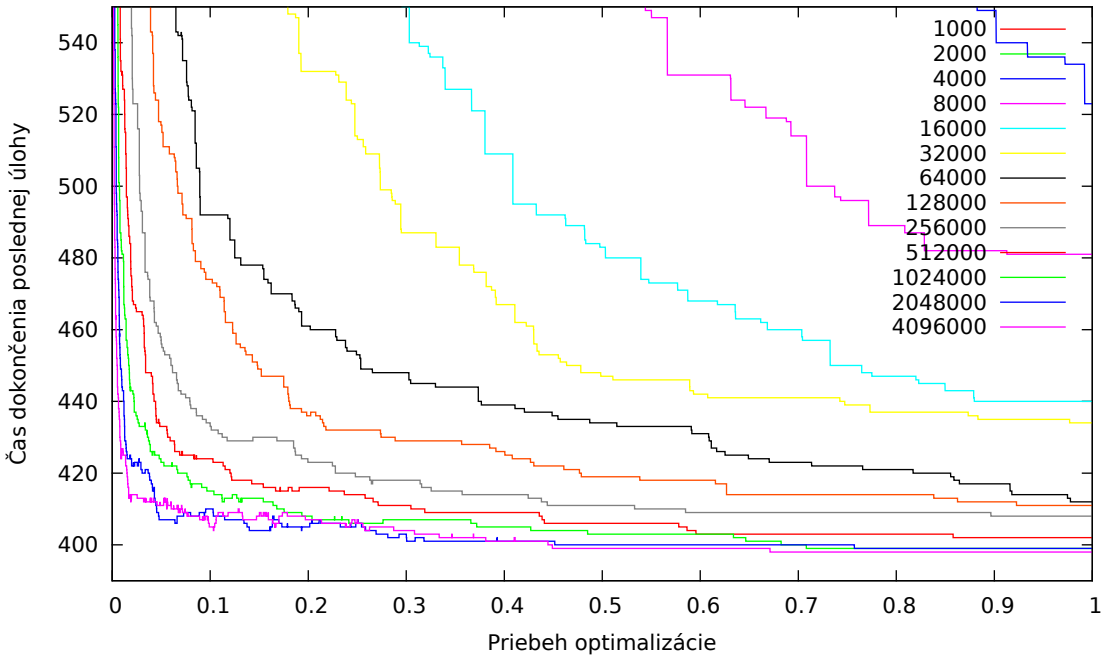
Obr. 5.1: Porovnanie CPU a GPU



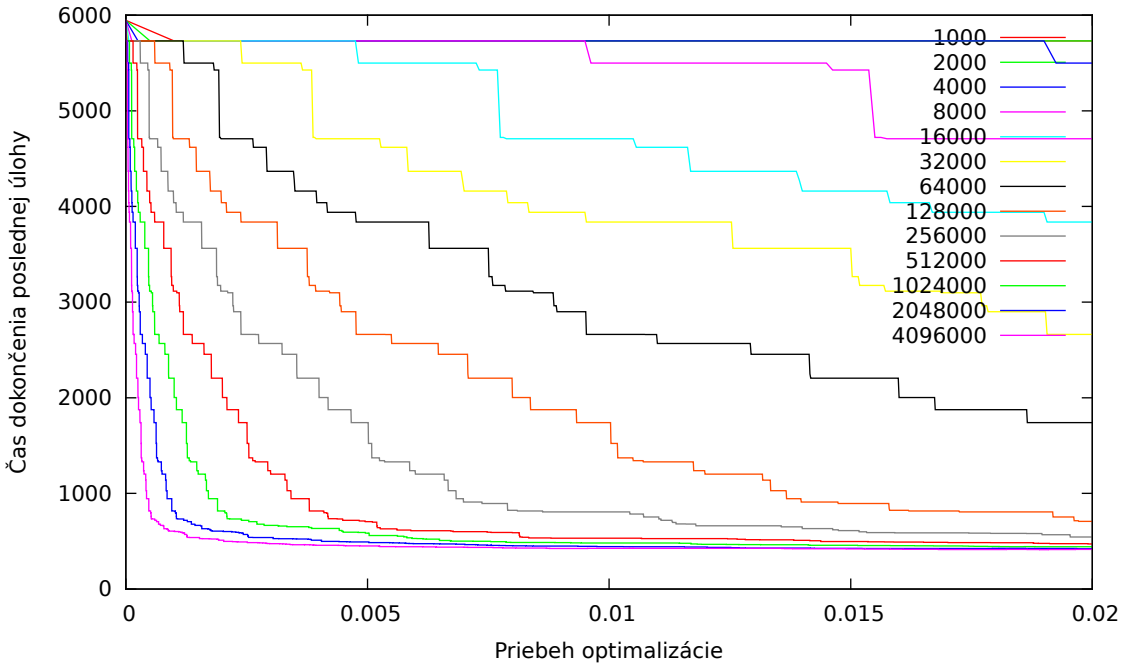
Obr. 5.2: Priebeh optimalizácie pre 60000 iterácií



Obr. 5.3: Priebeh optimalizácie podľa počtu iterácií



Obr. 5.4: Priebeh optimalizácie podľa počtu iterácií – detail 1



Obr. 5.5: Priebeh optimalizácie podľa počtu iterácií – detail 2

Záver

Diplomová práca je zameraná na problematiku plánovania úloh v gridovom prostredí. Jej cieľom je na základe analýzy modelov plánovania úloh v gridoch a cloudoch navrhnúť, implementovať a overiť model plánovania úloh v gridoch a cloudoch pomocou GPGPU.

Počas vývoja simulátora plánovania úloh a optimalizačného modelu bola overená vhodnosť použitia pokročilých plánovacích algoritmov pre plánovanie úloh v gridovom prostredí. Ako nástroj zvoleného riešenia bol vybratý centralizovaný plánovač, pretože na grafickej karte je najvhodnejšie spracovanie veľkého množstva dát. Pri návrhu plánovača bolo zohľadnené, že gridové zdroje sú len výpočtové zdroje, všetky úlohy majú tiež len výpočtový charakter a zohľadňuje sa prenos súborov medzi úlohami. Uvažuje sa len s nepreemptívnymi úlohami. Dĺžka výpočtu úloh sa počas rozvrhovania nemení, je známa dopredu.

Pre návrh optimalizačného modelu bol zvolený algoritmus simulovaného žíhania s cieľom vyhnúť sa pri optimalizácii uviaznutiu v lokálnom extréme. Model bol overovaný pri počte uzlov od 32 po 704. Zlepšenie implementácie na GPGPU sa začalo výrazne prejavovať od počtu 512 uzlov a so zvyšujúcim počtom uzlov sa zrýchlenie ďalej zvyšuje. Z toho vyplýva, že implementácia na GPGPU má význam pri riešení zložitých problémov. Perspektívu ďalšieho zrýchlenia je možné predpokladať v úprave programu optimalizáciou kernelov a paralelizáciou na viacerých grafických kartách a počítačoch.

Zoznam bibliografických odkazov

[1] NVIDIA, NVIDIA CUDA C Programming Guide version 3.0. Dostupné na internete: http://developer.nvidia.com/nvidia-gpu-computing-documentation .

[2] Chang R. S., Lin Ch. Y., Lin Ch. F.: "An Adaptive Scoring Job Scheduling alhorithm for grid computing". Information Sciences 207 (2012) 79-89. Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan.

[3] Kmuníček J.: "Gridy jako klíčový fenomén informačních technologií nového tisíciletí." Inflow: information journal [online]. 2008, roč. 1, č. 1. Dostupný z WWW: http://www.inflow.cz/gridy-jako-klicovy-fenomen-informacnich-technologii-noveho-tisicileti. ISSN 1802-9736.

[4] Povinský M., Škrinárová J.: "Simulátor plánovania úloh v gride" ŠVK UMB Banská Bystrica 2013

[5] Važan P., Pauliček R.: "Hodnotenie algoritmov simulačnej optimalizácie vo WITNESS" STU Trnava, 2010. Dostupný z WWW: http://www2.humusoft.cz/www/papers/witkonf10/witness2010-vazan.pdf

[6] Povinský M. 2012. Optimalizácia iregulárnych topológií prepojovacích sietí s využitím GPGPU: bakalárska práca. Banská Bystrica: FPV UMB, 2012. 55 s.

[7] Foster, I., Kesselman, C.: The Grid2: Blueprint for a New Computing Infrastructure (The Elsevier Series in Grid Computing), Morgan Kaufmann Publihers, 2003.

[8] Zelinka, I.: Biologicky inspirované výpočty: evoluční algoritmy. Dostupný z WWW: http://arg.vsb.cz/data/Vyuka/02%20BIV%20Evoluce%20-%20Uvod.pdf

[9] Klusáček, D.: Plánování úloh v paralelním a distribuovaném prostředí: diplomová práce. Brno: MUNI, 2006. 47s.

[10] Jerz V.: Simulačné a optimalizačné modely – základný nástroj na zlepšovanie procesov: Posterus. 2010. Dostupný z WWW: http://www.posterus.sk/?p=9164&output=pdf ISSN 1338-0087

[11] Mach M.: Evolučné algoritmy. Košice: ELFA, 2009 237 s. ISBN 978-80-8086-123-0

Univerzita Mateja Bela v Banskej Bystrici

Fakulta prírodných vied

Optimalizácia plánovania úloh v gridoch a cloudoch pomocou GPGPU

diplomová práca – zdrojový kód

Príloha A

Banská Bystrica, 2014 Bc. Michal Povinský

Listings

Listing 1: main.h

```
#ifndef MAIN_H
#define MAIN_H
#include "matrix.h"
#define M(x) (mtxs+(x))

extern int nodecount;
extern int jobcount;
extern int filecount;

enum mtxs {
  NODE_LATENCY,
  CPU_SPEED,
  JOB_LENGTH,
  FILE_SIZE,
  FILE_MIRRORS,
  JOB_FILES,
  FILE_GENBY,
  NODE_BANDWIDTH,
  NODEJOBS,
  NODEJOBCOUNT,
  NODECOMPLETEDJOBS,
  NODELASTJOBFINISH,
  NODEFILEDOWNLOAD,
  JOBDEP,
  JOBFILESLIST,
  JOBFILESCOUNT,
  NODEJOBSBEST,
  NODEJOBCOUNTBEST,
  NODEEDOWNLOADCOUNT,
  NODEEDOWNLOADTIMES,
  NODEFILEDOWNLOADFOR,
  JOBFINISH,
  KERNEL_STATUS
};

enum kernelstatus {
  KS_NODE_NOT_FINISHED,
  KS_JOB_IMPROVED,
  KS_JOB_DONE,
  KS_DEBUG,
  KS_MATRIX_MAX
};

#define KS_COUNT 5

extern struct matrix mtxs[];

void tempcheck();

#endif
```

Listing 2: matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H
```

```
#include <stdio.h>
#include <assert.h>
#include <stdint.h>

struct matrix {
  int x, y;
  int *data;
  int *cuda_data;
};

static inline int *mtx(struct matrix *m, int x, int y) {
  assert(x >= 0 && y >= 0 && x < m->x && y < m->y);
  return (m->data) + x + (y * m->x);
};

#define MTX(m,x,y) (*(mtx((m),(x),(y))))

void mtx_init(struct matrix *m, int x, int y);
void mtx_rand(struct matrix *m, int min, int max);
void mtx_setrand(struct matrix *m, int perc, int value);
void mtx_set(struct matrix *m, int value);
void mtx_sym_diag_min(struct matrix *m);
void mtx_fw(struct matrix *m);
void mtx_print(struct matrix *m, FILE *f);
void mtx_read(struct matrix *m, FILE *f);
void mtx_boolinvert(struct matrix *m);
void mtx_copy(struct matrix *d, struct matrix *s);
uint64_t mtx_sum(struct matrix *m);

#endif
```

Listing 3: cuda.h

```
#ifndef CUDA_H
#define CUDA_H
#include "matrix.h"
void mtx_fw_gpu(struct matrix *m);
int opt_sim_gpu();
void opt_sim_gpu_a();
#endif
```

Listing 4: main.c

```
#define __STDC_FORMAT_MACROS
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <assert.h>
#include <limits.h>
#include <time.h>
#include <unistd.h>
#include <math.h>

#include "main.h"
```

```
#include "matrix.h"
#include "sfmt/SFMT.h"

#define E(m,x,y) MTX(M(m),x,y)

#ifdef CUDA
//#define CUDA_OPT
#endif

int maxiter = 60000; //was 1000
int extrahcli = 10;
double tempexp = 1;
int tempstart = 0; // 250
int makegraphs = 1;
double temp_maxmult = 1;

char *mtxnames[] = {
  "node_latency", "cpu_speed", "job_length", "file_size",
     "file_mirrors",
  "job_files", "file_genby", "node_bandwidth",
  "_nodejobs", "_nodejobcount", "_nodecompletedjobs",
     "_nodelastjobfinish",
  "_nodefiledownload", "_jobdep", "_job_files_list",
  "_job_files_count", "_nodejobs_best", "_nodejobcount_best",
  "_node_e_download_count", "_node_e_download_times",
     "_nodefiledownloadfor",
  "_kernel_status", "_job_finish",
  NULL };

int nodecount;
int jobcount;
int filecount;

FILE *logfile = NULL;

struct matrix mtxs[sizeof(mtxnames)/sizeof(*mtxnames)];

int nvsleep = 0;

void tempcheck()
{
  char *path = "/sys/class/hwmon/hwmon0/temp1_input";
  int maxtemp = 65*1000;
  static int intg = 0;
  FILE *f;
  int t, tc, tg;

  f = fopen(path, "r");
  fscanf(f, "%d", &tc);
  fclose(f);

  while (1) {
    f = popen("./getgputemp", "r");
    fscanf(f, "%d", &tg);
    //tg -= 5;
```

```
    //tg *= 1000;
    fclose(f);
    if (tg < 75) break;
    nvsleep++;
    printf("nvsleep %d\n", nvsleep);
    sleep(1);
  }

  //t = tc > tg ? tc : tg;
  t = tc;

  int tempdif = t - maxtemp;
  intg += tempdif / 10;
  if (intg < 0) intg = 0;
  int slp = intg + 2 * tempdif;
  printf("TMPCHK cpu=%d gpu=%d mt=%d t=%d td=%d in=%d slp=%d\n", tc,
      tg, maxtemp, t, tempdif, intg, slp);

  //if (t < maxtemp) break;
  //printf("<T %d>", t/1000); fflush(stdout);
  //usleep(200*1000);
  if (slp > 0) usleep(slp);
}

char *strtrim(char *s)
{
  char *s2;
  while (*s && isspace(*s)) s++;
  s2 = s;
  while (isgraph(*s)) s++;
  *s = 0;
  return s2;
}

int read_named_matrix(FILE *f)
{
  char buf[64], *tbuf;
  int i;
  fgets(buf, sizeof(buf), f);
  if (feof(f)) {
    return 0;
  }
  tbuf = strtrim(buf);
  for (i=0; mtxnames[i]; i++)
    if (!strcmp(tbuf, mtxnames[i]))
      break;
  if (!mtxnames[i]) {
    fprintf(stderr, "unknown matrix '%s'\n", tbuf);
    exit(1);
  }
  fprintf(stderr, "loading matrix '%s'\n", tbuf);
  mtx_read(mtxs+i, f);
  printf("mtx %dx%d\n", mtxs[i].x, mtxs[i].y);
  return 1;
}
```

```
void graph_jobdep()
{
  int i, j;
  FILE *f = fopen("graph/jobdep.dot", "w");
  fprintf(f, "digraph {\n");
  //fprintf(f, "graph [aspect=\"2,100\" ratio=compress]\n");
  fprintf(f, "graph [mclimit=10]\n");
  for (i=0; i<jobcount; i++) {
    for (j=0; j<jobcount; j++) {
      if (E(JOBDEP, j, i)) {
        fprintf(f, "%d -> %d\n", i, j);
      }
    }
  }
  fprintf(f, "}\n");
  fclose(f);
}

void makejobdep()
{
  int i, j;
  mtx_set(M(JOBDEP), 1);
  for (i=0; i<jobcount; i++) {
    for (j=0; j<filecount; j++) {
      if (E(JOB_FILES, i, j) && E(FILE_GENBY, j, 0) >= 0) {
        E(JOBDEP, i, E(FILE_GENBY, j, 0)) = 0;
      }
    }
  }
  #ifdef CUDA
  mtx_fw_gpu(M(JOBDEP));
  #else
  mtx_fw(M(JOBDEP));
  #endif
  mtx_boolinvert(M(JOBDEP));
  //mtx_print(M(JOBDEP), stdout);
  for (i=0; i<jobcount; i++) {
    if (E(JOBDEP, i, i)) {
      fprintf(stderr, "Job %d depends on itself\n", i);
      abort();
    }
  }
}

void makejobfileslist()
{
  int j, f, i;
  mtx_set(M(JOBFILESLIST), -1);
  mtx_set(M(JOBFILESCOUNT), 0);
  for (j=0; j<jobcount; j++) {
    i = 0;
    for (f=0; f<filecount; f++) {
      if (E(JOB_FILES, j, f))
        E(JOBFILESLIST, j, i++) = f;
```

```
    }
    E(JOBFILESCOUNT, j, 0) = i;
  }
  //mtx_print(M(JOBFILESLIST), stdout);
  //mtx_print(M(JOBFILESCOUNT), stdout);
}

void opt_init()
{
  nodecount = mtxs[NODE_LATENCY].x;
  jobcount = mtxs[JOB_LENGTH].x;
  filecount = mtxs[FILE_SIZE].x;

  printf("nodecount %d jobcount %d filecount %d\n", nodecount,
      jobcount, filecount);

  mtx_init(M(NODEJOBS), nodecount, jobcount);
  mtx_init(M(NODEJOBCOUNT), nodecount, 1);
  mtx_init(M(NODECOMPLETEDJOBS), nodecount, 1);
  mtx_init(M(NODELASTJOBFINISH), nodecount, 1);
  mtx_init(M(NODEFILEDOWNLOAD), nodecount, filecount);
  mtx_init(M(JOBDEP), jobcount, jobcount);
  mtx_init(M(JOBFILESLIST), jobcount, filecount);
  mtx_init(M(JOBFILESCOUNT), jobcount, 1);
  mtx_init(M(NODEJOBSBEST), nodecount, jobcount);
  mtx_init(M(NODEJOBCOUNTBEST), nodecount, 1);
  mtx_init(M(NODEEDOWNLOADCOUNT), nodecount, 1);
  mtx_init(M(NODEEDOWNLOADTIMES), nodecount, filecount);
  mtx_init(M(NODEFILEDOWNLOADFOR), nodecount, filecount);
  mtx_init(M(JOBFINISH), jobcount, 1);

  #ifdef CUDA
    mtx_init(M(KERNEL_STATUS), KS_COUNT, 1);
  #endif

  makejobdep();
  makejobfileslist();

  if (makegraphs) graph_jobdep();

  init_gen_rand(12345);
}

void assign_jobs_randomly()
{
  int i, n, js;
  for (i = 0; i<jobcount; i++) {
    n = gen_rand32() % nodecount;
    js = MTX(M(NODEJOBCOUNT), n, 0)++;
    MTX(M(NODEJOBS), n, js) = i;
    //printf("Job %d node %d count %d\n", i, n, js+1);
  }
}

int job_lengthonnode(int n, int j)
```

```
{
  int jl = MTX(M(JOB_LENGTH), j, 0);
  int cs = MTX(M(CPU_SPEED), n, 0);
  //printf("job %d on %d = %d/%d=%d\n", j, n, jl, cs, (jl+cs-1)/cs);
  return (jl+cs-1)/cs;
}

int job_filetransfertime(int n1, int n2, int file)
{
  //printf("job_filetransfertime %d %d %d\n", n1, n2, file);
  int latency = E(NODE_LATENCY, n1, n2);
  //printf("latency is %d\n", latency);
  int bandwidth = E(NODE_BANDWIDTH, n1, n2); /* todo */
  int size = MTX(M(FILE_SIZE), file, 0);
  //printf("size is %d\n", size);
  return latency + size / bandwidth;
}

int job_filetransferfinishtime(int n1, int n2, int file, int dlidx)
{
  int createtime = E(FILE_MIRRORS, file, n1);
  int dest_idle = dlidx < 0 ? 0 : E(NODEEDOWNLOADTIMES, n2, dlidx);
  int downloadstart = createtime < dest_idle ? dest_idle : createtime;
  assert(createtime >= 0);
  //printf("created at %d idle at %d\n", createtime, dest_idle);
  //printf("tranfer from %d would start at %d\n", n1, downloadstart);
  return downloadstart + job_filetransfertime(n1, n2, file);
}

int job_selecttransfernode(int dn, int file, int dlidx)
{
  int n, sn = -1, otime = INT_MAX, time;
  for (n=0; n<nodecount; n++) {
    if (E(FILE_MIRRORS, file, n) < 0) {
      //printf("node %d does not have file %d\n", n, file);
      continue;
    }
    time = job_filetransferfinishtime(n, dn, file, dlidx);
    //printf("can download file %d from %d at %d\n", file, n, time);
    if (time < otime) {
      otime = time;
      sn = n;
    }
  }
  return sn;
}
//old 21528
int opt_sim()
{
  int n, j, f, dn, fi, maxtime = 0, ji;
  int this_file, all_files, jobstart, jobwait, jobfinish, improving;
  int node_not_finished = 1, job_done = 1, job_improved, file_download;
  int c_lastjobfinish, prev_last;

  mtx_set(M(NODECOMPLETEDJOBS), 0);
```

```
mtx_set(M(NODELASTJOBFINISH), 0);
mtx_set(M(NODEFILEDOWNLOAD), -1);
mtx_set(M(NODEEDOWNLOADCOUNT), 0);
mtx_set(M(NODEEDOWNLOADTIMES), -1);
mtx_set(M(NODEFILEDOWNLOADFOR), -1);
//mtx_print(M(NODEJOBS), stdout);

while (node_not_finished || job_improved) {
  //printf("nf %d ji %d jd %d\n", node_not_finished, job_improved,
      job_done);
  if (!job_done && node_not_finished) {
    //mtx_print(M(NODEJOBCOUNT), stdout);
    //mtx_print(M(NODEJOBS), stdout);
    //printf("deadlocked!\n");
    //exit(1);
    return -1;
  }
  node_not_finished = 0;
  job_done = 0;
  job_improved = 0;
  for (n=0; n<nodecount; n++) {
    //if (n==0) printf("*** node: %d\n", n);
    prev_last = E(NODELASTJOBFINISH, n, 0);
    c_lastjobfinish = 0;
    file_download = 0;
    for (ji=0; ji<MTX(M(NODEJOBCOUNT),n,0); ji++) {
      improving = (ji < E(NODECOMPLETEDJOBS, n, 0));
      j = E(NODEJOBS, n, ji);
      //printf("job %d[%d] -> %d\n", n, ji, j);
      if (improving) {
        //printf("improving job %d\n", j);
        //continue;
      }
      //printf("job needs %d files\n", E(JOBFILESCOUNT, j, 0));
      all_files = 0;
      for (fi=0; fi<E(JOBFILESCOUNT, j, 0); fi++) {
        f = E(JOBFILESLIST, j, fi);
        //if (n==0) printf("job %d needs file %d\n", j, f);
        if ((this_file = E(FILE_MIRRORS, f, n)) >= 0) {
          if (E(NODEFILEDOWNLOADFOR, n, f) != j) {
            //if (n==0) printf("file %d for job %d already present
                since %d\n", f, j, this_file);
            continue;
          } else {
            //if (n==0) printf("file [%d] %d for job %d downloaded at
                %d improving\n", file_download, f, j, this_file);
            //dn = job_selecttransfernode(n, f, file_download - 1);
          }
        }
          //printf("file not present, downloading\n");
          //printf("download_idx %d\n", file_download);
          //mtx_print(M(NODEEDOWNLOADTIMES), stdout);
          dn = job_selecttransfernode(n, f, file_download - 1);
          if (dn < 0) {
            //printf("cannot download file %d for job %d on node %d
```

```
                genby %d\n", f, j, n, E(FILE_GENBY, f, 0));
              node_not_finished = 1;
              goto nextnode;
            }
            //printf("will transfer from %d\n", n);
            this_file = job_filetransferfinishtime(dn, n, f,
                file_download - 1);
            //if (n==0 && ji==2) printf("file %d[%d] for job %d
                downloaded from %d at %d\n", f, fi, j, dn, this_file);
            if (E(NODEEDOWNLOADTIMES, n, file_download) > this_file) {
              // printf("file download improved %d -> %d\n",
                  E(NODEEDOWNLOADTIMES, n, file_download), this_file);
              job_improved = 1;
            }
            E(NODEEDOWNLOADTIMES, n, file_download) = this_file;
            E(FILE_MIRRORS, f, n) = this_file;
            E(NODEFILEDOWNLOADFOR, n, f) = j;

          if (this_file > all_files)
            all_files = this_file;
          file_download++;
        }
        //if (improving) continue; //fixme
        jobstart = c_lastjobfinish;
        jobwait = all_files > jobstart ? all_files - jobstart : 0;
        if (all_files > jobstart)
          jobstart = all_files;
        jobfinish = jobstart + job_lengthonnode(n, j);
        c_lastjobfinish = jobfinish;
        MTX(M(NODELASTJOBFINISH), n, 0) = c_lastjobfinish;
        MTX(M(JOBFINISH), j, 0) = jobfinish;
        if (!improving) {
          MTX(M(NODECOMPLETEDJOBS), n, 0) ++;
          job_done++;
        }
        for (f=0; f<filecount; f++) {
          if (E(FILE_GENBY, f, 0) == j) {
            E(FILE_MIRRORS, f, n) = jobfinish;
            E(NODEFILEDOWNLOADFOR, n, f) = j;
            //printf("job %d created file %d on %d at %d\n", j, f, n,
                jobfinish);
          }
        }
        //job_done = 1;
        //if (n==0) printf("job %d[%d] waits for %d, starts at %d, runs
            for %d, finishes at %d\n", j, ji, jobwait, jobstart,
            jobfinish-jobstart, jobfinish);
      }
      nextnode: {}
      //printf("last prev=%d new=%d\n", prev_last, E(NODELASTJOBFINISH,
          n, 0));
    }
  }
  for (n=0; n<nodecount; n++) {
    //printf("node %d completed %d jobs in %d\n", n,
```

```
        MTX(M(NODECOMPLETEDJOBS), n, 0), MTX(M(NODELASTJOBFINISH), n,
        0));
    if (E(NODELASTJOBFINISH, n, 0) > maxtime)
      maxtime = E(NODELASTJOBFINISH, n, 0);
  }
  //printf("jobs completed in %d\n", maxtime);
  return maxtime;
}

int do_change()
{
  //mtx_print(M(NODEJOBCOUNT), stdout);
  //mtx_print(M(NODEJOBS), stdout);

  //printf("moving job!\n");
  //move a job
  int srcnode = gen_rand32() % nodecount;
  int dstnode = gen_rand32() % nodecount;
  //if (gen_rand32() % 100 < 50) dstnode = srcnode;
  int srcjc = E(NODEJOBCOUNT, srcnode, 0);
  int dstjc = E(NODEJOBCOUNT, dstnode, 0);
  if (srcnode==dstnode) dstjc--;
  if (!srcjc)
    return 0;
  //printf("jc %d %d\n", srcjc, dstjc);
  int srcjob = gen_rand32() % srcjc;
  int dstjob = gen_rand32() % (dstjc + 1);
  //printf("job %d %d\n", srcjc, dstjc);
  int job = E(NODEJOBS, srcnode, srcjob);
  int i;
  //printf("job %d %d@%d -> %d@%d\n", job, srcjob, srcnode, dstjob,
      dstnode);
  for (i=srcjob; i<srcjc - 1; i++)
    E(NODEJOBS, srcnode, i) = E(NODEJOBS, srcnode, i+1);
  E(NODEJOBCOUNT, srcnode, 0)--;
  E(NODEJOBS, srcnode, srcjc-1) = 0;
  for (i=dstjc; i>dstjob; i--)
    E(NODEJOBS, dstnode, i) = E(NODEJOBS, dstnode, i-1);
  E(NODEJOBCOUNT, dstnode, 0)++;
  E(NODEJOBS, dstnode, dstjob) = job;
  //printf("moved\n");

  int oj;
  //mtx_print(M(NODEJOBS), stdout);
  for (i=0; i<dstjob; i++) {
    oj = E(NODEJOBS, dstnode, i);
    //if (dstnode==1) printf("check %d %d\n", oj, job);
    if (E(JOBDEP, oj, job)) {
      //printf("baddep %d %d\n", oj, job);
      return 0;
    }
  }
  for (i=dstjob; i<dstjc + 1; i++) {
    oj = E(NODEJOBS, dstnode, i);
    //if (dstnode==1) printf("check %d %d[%d@%d]\n", job, oj, i,
```

```
        dstnode);
    if (E(JOBDEP, job, oj)) {
      //printf("baddep %d %d\n", job, oj);
      return 0;
    }
  }
  //mtx_print(M(NODEJOBCOUNT), stdout);
  //mtx_print(M(NODEJOBS), stdout);
  return 1;
}

int timer()
{
  static struct timespec told;
  struct timespec tnew;
  int t;
  clock_gettime(CLOCK_MONOTONIC, &tnew);
  t = (tnew.tv_sec - told.tv_sec) * 1000 + (tnew.tv_nsec -
     told.tv_nsec) / 1000000;
  told = tnew;
  return t;
}

void reset_filemirrors()
{
  int f,n;
  for (n=0; n<nodecount; n++)
    for (f=0; f<filecount; f++)
      if (E(FILE_MIRRORS, f, n) > 0) E(FILE_MIRRORS, f, n) = -1;
}

int opt_sim_c(uint64_t *jobtotaltime)
{
  int tim;
  int rtc, rtg;
  int check = 0;

  #ifdef CUDA_OPT
  int timc;
  if (check) {
    timer();
  }
  reset_filemirrors();
  tim = opt_sim_gpu();
  if (check) {
    rtg = timer();
    reset_filemirrors();
    timc = opt_sim();
    rtc = timer();
  }
  opt_sim_gpu_a();
  if (check && timc != tim) {
    printf("time: CPU %d GPU %d\n", timc, tim);
    printf("rtime: CPU %d GPU %d spd %f\n", rtc, rtg, (double)rtc/rtg);
    sleep(3);
```

```
  }
  #else
  tempcheck();
  tim = opt_sim();
  #endif
  *jobtotaltime = mtx_sum(M(JOBFINISH));
  return tim;
}

void swap(int *a, int *b)
{
  int t;
  t = *a;
  *a = *b;
  *b = t;
}

void rand_file_downloads()
{
  int j, fi;
  for (j=0; j<jobcount; j++) {
    for (fi=1; fi<E(JOBFILESCOUNT, j, 0); fi++) {
      //printf("SWF %d ", fi);
      if (gen_rand32()%400 == 0) {
        swap(&E(JOBFILESLIST, j, fi), &E(JOBFILESLIST, j, fi-1));
        //printf("SWP %d %d\n", j, fi);
      }
    }
  }
}

uint64_t score(int max, int total)
{
  return max * jobcount * temp_maxmult + total;
}

double gettemp(int iter, int maxiter)
{
  double saratio = 1;
  double progr = (double)iter/(maxiter*saratio);
  if (progr > 1) return 0;
  //printf("%f\n", progr);
  //return tempstart * (maxiter - iter) / maxiter;
  progr = pow(progr, tempexp);
  return tempstart * (1-progr);
}

void opt()
{
  int best, last, i, iter, depok, code, accept, lastimprv = 0;
  double temp, acceptp, roll;
  uint64_t bestt, lastt, bests, lasts;
  int64_t deltas;
  assign_jobs_randomly();
  mtx_copy(M(NODEJOBSBEST), M(NODEJOBS));
```

```
mtx_copy(M(NODEJOBCOUNTBEST), M(NODEJOBCOUNT));

//printf("nodejobs:\n"); mtx_print(M(NODEJOBS), stdout);
//printf("jobfilescount:\n"); mtx_print(M(JOBFILESCOUNT), stdout);

//printf("file_mirrors:\n"); mtx_print(M(FILE_MIRRORS), stdout);

best = opt_sim_c(&bestt);
for (iter = 0; iter < maxiter || lastimprv < extrahcli;) {
  depok = 1;
  int changes = gen_rand32() % 4 + 1;
  for (i=0; i<changes; i++)
    depok = depok && do_change();
  //printf("depok %d\n", depok);

  rand_file_downloads();
  //printf("jobfileslist:\n"); mtx_print(M(JOBFILESLIST), stdout);

  //printf("%8d ", iter);

  if (!depok) {
    mtx_copy(M(NODEJOBS), M(NODEJOBSBEST));
    mtx_copy(M(NODEJOBCOUNT), M(NODEJOBCOUNTBEST));
    printf("d %d\n", best);
    continue;
  }

  last = opt_sim_c(&lastt);

  if (last == -1) {
    mtx_copy(M(NODEJOBS), M(NODEJOBSBEST));
    mtx_copy(M(NODEJOBCOUNT), M(NODEJOBCOUNTBEST));
    printf("D %d\n", best);
    continue;
  }
  iter++;
  int isworse = last > best || (last == best && lastt > bestt);
  int isbetter = last < best || (last == best && lastt < bestt);
  bests = score(best, bestt);
  lasts = score(last, lastt);
  deltas = lasts - bests;
  temp = gettemp(iter, maxiter);
  if (temp > 0) {
    acceptp = deltas < 0 ? 1 : exp(-deltas / temp);
    roll = (double)gen_rand32()/UINT_MAX;
    accept = (roll < acceptp);
  } else {
    accept = !isworse;
  }


  if (!accept) {
    mtx_copy(M(NODEJOBS), M(NODEJOBSBEST));
    mtx_copy(M(NODEJOBCOUNT), M(NODEJOBCOUNTBEST));
    code = '-';
```

```
    } else {
      best = last;
      bestt = lastt;
      mtx_copy(M(NODEJOBSBEST), M(NODEJOBS));
      mtx_copy(M(NODEJOBCOUNTBEST), M(NODEJOBCOUNT));
      code = '+';
    }

    if (isbetter) {
      lastimprv = 0;
    } else {
      lastimprv++;
    }

    printf("%c %6d/%d %d %d | %" PRIu64 " %" PRIu64 " | %" PRIu64 " %"
        PRIu64 " %8" PRIi64 " | %3.0f%% t=%f A%c %f LI=%d\n",
      code, iter, maxiter, best, last, bestt, lastt, bests, lasts,
          deltas, acceptp * 100, temp, accept ? '+' : '-', roll,
          lastimprv);
    fprintf(logfile, "%d %d %" PRIu64 " %" PRIu64 " %" PRIu64 " %"
        PRIu64 " %f\n", best, last, bestt, lastt, bests, lasts, temp);
  }
}

int main(int argc, char **argv)
{
  int i;
  if (argc > 1) {
    stdin = fopen(argv[1], "r");
    if (!stdin) perror("fopen");
  }
  for (i=0; mtxnames[i]; i++)
    mtxs[i].data = NULL;
  while (read_named_matrix(stdin));
  for (i=0; mtxnames[i]; i++) {
    if (mtxnames[i][0] != '_' && !mtxs[i].data) {
      fprintf(stderr, "Matrix %s missing\n", mtxnames[i]);
      exit(1);
    }
  }

  opt_init();

  logfile = fopen("main.log", "w");

  opt();

  fclose(logfile);

  return 0;
}
```

Listing 5: matrix.c

```
#include <stdlib.h>
#include <assert.h>
```

```
#include <limits.h>
#include <string.h>
#include <stdio.h>

#include "sfmt/SFMT.h"
#include "matrix.h"
#include "main.h"

void mtx_init(struct matrix *m, int x, int y)
{
  m->x = x;
  m->y = y;
  m->data = malloc(sizeof(m->data[0]) * x * y);
  m->cuda_data = NULL;
}

void mtx_rand(struct matrix *m, int min, int max)
{
  int i;
  for (i=0; i<m->x*m->y; i++) {
    m->data[i] = min + gen_rand32() % (max-min+1);
  }
}

void mtx_setrand(struct matrix *m, int perc, int value)
{
  int i;
  for (i=0; i<m->x*m->y; i++)
    if (gen_rand32() % 100 < perc)
      m->data[i] = value;
}

void mtx_set(struct matrix *m, int value)
{
  int i;
  for (i=0; i<m->x*m->y; i++)
    m->data[i] = value;
}

void mtx_sym_diag_min(struct matrix *m)
{
  assert(m->x == m->y);
  int x,y;
  for (x=0; x<m->x; x++)
    for (y=0; y<m->y; y++)
      if (MTX(m,x,y) > MTX(m,y,x))
        MTX(m,x,y) = MTX(m,y,x);
}

void mtx_fw(struct matrix *m)
{
  assert(m->x == m->y);
  int i,j,k;
  for (i=0; i<m->x; i++) {
    fprintf(stderr, "%d/%d\n", i, m->x);
```

```
    tempcheck();
    for (j=0; j<m->x; j++)
      for (k=0; k<m->x; k++)
        if (MTX(m,j,i) < INT_MAX && MTX(m,i,k) < INT_MAX &&
          MTX(m,j,k) > MTX(m,j,i) + MTX(m,i,k))
          MTX(m,j,k) = MTX(m,j,i) + MTX(m,i,k);
  }
}

void mtx_print(struct matrix *m, FILE *f)
{
  int i;
  fprintf(f, "matrix %d %d\n", m->x, m->y);
  for (i=0; i<m->x*m->y; i++) {
    fprintf(f, "%d%c", m->data[i], i%m->x==m->x-1 ? '\n' : ' ');
  }
}

void mtx_read(struct matrix *m, FILE *f)
{
  int x, y, r, i;
  r = fscanf(f, "matrix %d%d\n", &x, &y);
  assert(r==2);
  mtx_init(m, x, y);
  for (i=0; i < m->x * m->y; i++) {
    r = fscanf(f, "%d", m->data + i);
    assert(r==1);
  }
  fscanf(f, "\n");
}

void mtx_boolinvert(struct matrix *m)
{
  int i;
  for (i=0; i<m->x*m->y; i++) {
    m->data[i] = !m->data[i];
  }
}

void mtx_copy(struct matrix *d, struct matrix *s)
{
  assert(d->x == s->x);
  assert(d->y == s->y);
  memcpy(d->data, s->data, d->x * d->y * sizeof(int));
}

uint64_t mtx_sum(struct matrix *m)
{
  uint64_t s = 0;
  int x,y;
  for (x=0; x<m->x; x++)
    for (y=0; y<m->y; y++)
      s += MTX(m,x,y);
  return s;
}
```

Listing 6: genfiles.c

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "sfmt/SFMT.h"
#include "matrix.h"

static inline int min(x,y)
{
  if (x<y) return x;
  return y;
}

int main(int argc, char **argv)
{
  int nodes = 32;
  int mindist = 8;
  int maxdist = 32;
  int linkprob = 10;
  int jobcount = 32;
  int filecount = 32;
  int generatedfilepct = 50;
  int filesperjob = 8;
  int minbw = 100;
  int maxbw = 4000;

  if (argc >= 2) nodes = atoi(argv[1]);
  if (argc >= 3) jobcount = atoi(argv[2]);
  if (argc >= 4) filecount = atoi(argv[3]);

  fprintf(stderr, "n %d j %d f %d\n", nodes, jobcount, filecount);

  int i, j, k, t;

  init_gen_rand(12345);

  struct matrix m_distances;
  mtx_init(&m_distances, nodes, nodes);
  mtx_rand(&m_distances, mindist, maxdist);
  mtx_setrand(&m_distances, 100-linkprob, INT_MAX);
  mtx_sym_diag_min(&m_distances);
  mtx_fw(&m_distances);
  printf("node_latency\n");
  mtx_print(&m_distances, stdout);

  struct matrix m_cpuspeed;
  mtx_init(&m_cpuspeed, nodes, 1);
  mtx_rand(&m_cpuspeed, 1, 20);
  printf("cpu_speed\n");
  mtx_print(&m_cpuspeed, stdout);

//  struct matrix m_cpucount;
//  mtx_init(&m_cpucount, nodes, 1);
//  mtx_rand(&m_cpucount, 1, 64);
//  mtx_print(&m_cpucount, stdout);
```

```
struct matrix m_joblength;
mtx_init(&m_joblength, jobcount, 1);
mtx_rand(&m_joblength, 10, 1000);
printf("job_length\n");
mtx_print(&m_joblength, stdout);

struct matrix m_filesize;
mtx_init(&m_filesize, filecount, 1);
mtx_rand(&m_filesize, 1, 10000);
printf("file_size\n");
mtx_print(&m_filesize, stdout);

struct matrix m_filegenby;
mtx_init(&m_filegenby, filecount, 1);
mtx_rand(&m_filegenby, 0, jobcount-1);
mtx_setrand(&m_filegenby, 100-generatedfilepct, -1);
printf("file_genby\n");
mtx_print(&m_filegenby, stdout);

struct matrix m_jobfiles;
mtx_init(&m_jobfiles, jobcount, filecount);
mtx_set(&m_jobfiles, 0);
//mtx_setrand(&m_jobfiles, 1, 1);
for (i=0; i<jobcount; i++) {
  for (j=0; j<filesperjob; j++) {
    MTX(&m_jobfiles, i, gen_rand32() % filecount) = 1;
  }
}
for (i=0; i<jobcount; i++) {
  for (j=0; j<filecount; j++) {
    if (MTX(&m_filegenby, j, 0) >= i) {
      MTX(&m_jobfiles, i, j) = 0;
    }
  }
}
printf("job_files\n");
mtx_print(&m_jobfiles, stdout);

struct matrix m_filemirr;
mtx_init(&m_filemirr, filecount, nodes);
mtx_set(&m_filemirr, -1);
for (i = 0; i<filecount; i++) {
  if (MTX(&m_filegenby, i, 0) < 0)
    MTX(&m_filemirr, i, gen_rand32() % nodes) = 0;
}
printf("file_mirrors\n");
mtx_print(&m_filemirr, stdout);

struct matrix m_bandwidth;
mtx_init(&m_bandwidth, nodes, nodes);
mtx_rand(&m_bandwidth, minbw, maxbw);
mtx_setrand(&m_bandwidth, 100-linkprob, 1);
//mtx_print(&m_bandwidth, stdout);
for (i=0; i<nodes; i++) {
```

```
    for (j=0; j<nodes; j++) {
      for (k=0; k<nodes; k++) {
        t = min(MTX(&m_bandwidth, i, k), MTX(&m_bandwidth, k, j));
        if (MTX(&m_bandwidth, i, j) < t)
          MTX(&m_bandwidth, i, j) = t;
      }
    }
  }
  printf("node_bandwidth\n");
  mtx_print(&m_bandwidth, stdout);

  return 0;
}
```

Listing 7: cuda.cu

```
#include <assert.h>
#include <unistd.h>
extern "C" {
#include "matrix.h"
#include "cuda.h"
#include "main.h"
}

#define E(data,size,x,y) (data[x+size*y])
#define CD(mtx) (M(mtx)->cuda_data)

void errcheck() {
  if (cudaPeekAtLastError() == cudaSuccess) return;
  printf("err: %s\n", cudaGetErrorString(cudaPeekAtLastError()));
  abort();
}

void to_gpu(struct matrix *m)
{
  if (!m->cuda_data) {
    cudaMalloc(&m->cuda_data, m->x * m->y * sizeof(int));
    errcheck();
  }
  cudaMemcpy(m->cuda_data, m->data, sizeof(int) * m->x * m->y,
      cudaMemcpyHostToDevice);
  errcheck();
}

void to_host(struct matrix *m)
{
  cudaMemcpy(m->data, m->cuda_data, sizeof(int) * m->x * m->y,
      cudaMemcpyDeviceToHost);
  errcheck();
}

static __global__ void mtx_fw_k(int *data, int size, int i)
{
  int j = blockIdx.x * 16 + threadIdx.x;
  int k = blockIdx.y * 16 + threadIdx.y;
  if (j<size && k<size)
```

```
    if (E(data,size,j,k) > E(data,size,j,i) + E(data,size,i,k))
      E(data,size,j,k) = E(data,size,j,i) + E(data,size,i,k);
    //E(data,size,j,k) = min(E(data,size,j,i) + E(data,size,i,k),
        E(data,size,j,k));
}

void mtx_fw_gpu(struct matrix *m)
{
  int i;
  assert(m->x == m->y);
  dim3 tpb(16, 16);
  dim3 nb((m->x+15)/16, (m->x+15)/16);

  to_gpu(m);
  for (i=0; i<m->x; i++) {
    tempcheck();
    if (m->x > 255) printf("FW_%d/%d\n", i, m->x);
    mtx_fw_k<<<nb,tpb>>>(m->cuda_data, m->x, i);
    errcheck();
    cudaDeviceSynchronize();
    errcheck();
  }
  to_host(m);
}

static __global__ void mtx_max_g(int size, int *data, int *ks)
{
  int max = 0, i;
  for (i=threadIdx.x; i<size; i+=32) {
    if (max < data[i]) max = data[i];
  }
  atomicMax(ks + KS_MATRIX_MAX, max);
}

int mtx_max_gpu(struct matrix *m)
{
  int size = m->x * m->y;
  MTX(M(KERNEL_STATUS), KS_MATRIX_MAX, 0) = 0;
  to_gpu(M(KERNEL_STATUS));
  mtx_max_g<<<1, 32>>>(size, m->cuda_data, CD(KERNEL_STATUS));
  errcheck();
  to_host(M(KERNEL_STATUS));
  return MTX(M(KERNEL_STATUS), KS_MATRIX_MAX, 0);
}

static __device__ int job_lengthonnode_g(
  int nodecount, int jobcount, int filecount,
  int *joblength, int *cpuspeed,
  int n, int j)
{
  int jl = E(joblength, jobcount, j, 0);
  int cs = E(cpuspeed, nodecount, n, 0);
  return (jl+cs-1)/cs;
}
```

```
static __device__ int job_filetransfertime_g(
  int nodecount, int jobcount, int filecount,
  int *nodelatency, int *filesize, int *nodebandwidth,
  int n1, int n2, int file)
{
  int latency = E(nodelatency, nodecount, n1, n2);
  int bandwidth = E(nodebandwidth, nodecount, n1, n2);
  int size = E(filesize, filecount, file, 0);
  return latency + size / bandwidth;
}

static __device__ int job_filetransferfinishtime_g(
  int nodecount, int jobcount, int filecount,
  int *filemirrors, int *nodeedownloadtimes, int *nodelatency, int
      *filesize, int *nodebandwidth,
  int n1, int n2, int file, int dlidx)
{
  int createtime = E(filemirrors, filecount, file, n1);
  int dest_idle = dlidx < 0 ? 0 : E(nodeedownloadtimes, nodecount, n2,
      dlidx);
  int downloadstart = createtime < dest_idle ? dest_idle : createtime;
  return downloadstart + job_filetransfertime_g(nodecount, jobcount,
      filecount, nodelatency, filesize, nodebandwidth, n1, n2, file);
}

static __device__ int job_selecttransfernode_g(
  int nodecount, int jobcount, int filecount,
  int *filemirrors, int *nodeedownloadtimes, int *nodelatency, int
      *filesize, int *nodebandwidth,
  int dn, int file, int dlidx)
{
  int n, sn = -1, otime = INT_MAX, time;
  for (n=0; n<nodecount; n++) {
    if (E(filemirrors, filecount, file, n) < 0) {
      continue;
    }
    time = job_filetransferfinishtime_g(nodecount, jobcount, filecount,
        filemirrors, nodeedownloadtimes, nodelatency, filesize,
        nodebandwidth, n, dn, file, dlidx /* todo */);
    if (time < otime) {
      otime = time;
      sn = n;
    }
  }
  return sn;
}

#define SETDEBUG(x) { if (node == 0) kernelstatus[KS_DEBUG] = x; }

static __global__ void opt_sim_k(
  int nodecount, int jobcount, int filecount,
  int *nodelastjobfinish, int *nodejobcount, int *nodecompletedjobs,
      int *nodejobs, int *jobfilescount, int *jobfileslist, int
      *filemirrors, int *nodefiledownloadfor,
  int *nodeedownloadtimes, int *nodelatency, int *nodebandwidth, int
```

```
    *filesize, int *joblength, int *cpuspeed, int *filegenby, int
    *jobfinishtime, int *kernelstatus
  )
{
  int ji, fi, j, f, this_file, dn, job_improved = 0, jobstart,
    jobfinish, job_done = 0;
  //int jobwait;
  int node = blockIdx.x * blockDim.x + threadIdx.x;
  if (node >= nodecount) return;
  //int prev_last = E(nodelastjobfinish, nodecount, node, 0);
  int c_lastjobfinish = 0;
  int file_download = 0;
  for (ji = 0; ji<E(nodejobcount, nodecount, node, 0); ji++) {
    //SETDEBUG(1+ji)
    int improving = (ji < E(nodecompletedjobs, nodecount, node, 0));
    j = E(nodejobs, nodecount, node, ji);
    //if (improving) continue;
    int all_files = 0;
    for (fi = 0; fi < E(jobfilescount, jobcount, j, 0); fi++) {
      f = E(jobfileslist, jobcount, j, fi);
      if ((this_file = E(filemirrors, filecount, f, node)) >= 0) {

        if (E(nodefiledownloadfor, nodecount, node, f) != j) {

          continue;
        }
      }
      if (node==0 && ji==2 && fi==1) kernelstatus[KS_DEBUG] = -2;
      dn = job_selecttransfernode_g(nodecount, jobcount, filecount,
          filemirrors, nodeedownloadtimes, nodelatency, filesize,
          nodebandwidth, node, f, file_download - 1);
      if (dn < 0) {
        kernelstatus[KS_NODE_NOT_FINISHED] = 1;
        goto end;
      }
      this_file = job_filetransferfinishtime_g(nodecount, jobcount,
          filecount, filemirrors, nodeedownloadtimes, nodelatency,
          filesize, nodebandwidth, dn, node, f, file_download - 1);
      int file_oldtime = E(nodeedownloadtimes, nodecount, node,
          file_download);
      if (file_oldtime > this_file) {
        job_improved = 1;
      }
      //if (node==0 && ji==2 && fi==1) kernelstatus[KS_DEBUG] =
          this_file;
      if (file_oldtime != this_file) {
        E(nodeedownloadtimes, nodecount, node, file_download) =
            this_file;
        E(filemirrors, filecount, f, node) = this_file;
      }

      if (!improving) E(nodefiledownloadfor, nodecount, node, f) = j;

      if (this_file > all_files)
        all_files = this_file;
```

```
      file_download++;
    }
    //if (improving) continue;
    jobstart = c_lastjobfinish;
    //jobwait = all_files > jobstart ? all_files - jobstart : 0;
    if (all_files > jobstart)
      jobstart = all_files;
    jobfinish = jobstart + job_lengthonnode_g(nodecount, jobcount,
        filecount, joblength, cpuspeed, node, j);
    c_lastjobfinish = jobfinish;
    E(nodelastjobfinish, nodecount, node, 0) = c_lastjobfinish;
    E(jobfinishtime, jobcount, j, 0) = jobfinish;
    if (!improving) {
      E(nodecompletedjobs, nodecount, node, 0) ++;
      job_done = 1;
    }
    //if (node == 0 && ji == 2) kernelstatus[KS_DEBUG] = jobstart;
    for (f=0; f<filecount; f++) {
      if (E(filegenby, filecount, f, 0) == j) {
        E(filemirrors, filecount, f, node) = jobfinish;
        //E(nodefiledownloadfor, nodecount, node, f) = j; //not needed?
      }
    }
    //job_done = 1;
  }
  end:
  if (job_done) kernelstatus[KS_JOB_DONE] = 1;
  if (job_improved) kernelstatus[KS_JOB_IMPROVED] = 1;
}
#define CERR printf("err: %s\n",
    cudaGetErrorString(cudaPeekAtLastError()))
int opt_sim_gpu()
{
  static int kl = 0;
  int node_not_finished = 1, job_improved = 0, job_done = 1;

  mtx_set(M(NODECOMPLETEDJOBS), 0);
  to_gpu(M(NODECOMPLETEDJOBS));
  mtx_set(M(NODELASTJOBFINISH), 0);
  to_gpu(M(NODELASTJOBFINISH));
  mtx_set(M(NODEFILEDOWNLOAD), -1);
  to_gpu(M(NODEFILEDOWNLOAD));
  mtx_set(M(NODEEDOWNLOADCOUNT), 0);
  to_gpu(M(NODEEDOWNLOADCOUNT));
  mtx_set(M(NODEEDOWNLOADTIMES), -1);
  to_gpu(M(NODEEDOWNLOADTIMES));
  mtx_set(M(NODEFILEDOWNLOADFOR), -1);
  to_gpu(M(NODEEDOWNLOADTIMES));
  mtx_set(M(JOBFINISH), -1);
  to_gpu(M(JOBFINISH));

  to_gpu(M(FILE_MIRRORS));
  to_gpu(M(JOBFILESLIST));
  to_gpu(M(JOBFILESCOUNT));
  to_gpu(M(NODEJOBCOUNT));
```

```
  to_gpu(M(NODEJOBS));
  to_gpu(M(NODEFILEDOWNLOADFOR));
  to_gpu(M(NODE_LATENCY));
  to_gpu(M(NODE_BANDWIDTH));
  to_gpu(M(FILE_SIZE));
  to_gpu(M(JOB_LENGTH));
  to_gpu(M(CPU_SPEED));
  to_gpu(M(FILE_GENBY));

  while (node_not_finished || job_improved) {
    //node_not_finished = 0;
    if (!job_done && node_not_finished) {
      //printf("deadlocked\n");
      //exit(1);
      return -1;
    }

//  int *nodelastjobfinish, int *nodejobcount, int *nodecompletedjobs,
    int *nodejobs, int *jobfilescount, int *jobfileslist, int
    *filemirrors, int *nodefiledownloadfor,
//  int *nodeedownloadtimes, int *nodelatency, int *filesize, int
    *joblength, int *cpuspeed, int *filegenby
    mtx_set(M(KERNEL_STATUS), 0);
    M(KERNEL_STATUS)->data[KS_DEBUG] = -1;
    to_gpu(M(KERNEL_STATUS));
    int tpb = 16;
    //CERR;
    //printf("<<k %d>> ", kl++); fflush(stdout);

    /*to_host(M(FILE_MIRRORS));
    mtx_print(M(FILE_MIRRORS), stdout);*/

    tempcheck();

    opt_sim_k<<<(nodecount + tpb - 1) / tpb, tpb>>>(
      nodecount, jobcount, filecount,
      CD(NODELASTJOBFINISH), CD(NODEJOBCOUNT), CD(NODECOMPLETEDJOBS),
          CD(NODEJOBS),
      CD(JOBFILESCOUNT), CD(JOBFILESLIST), CD(FILE_MIRRORS),
          CD(NODEFILEDOWNLOADFOR),
      CD(NODEEDOWNLOADTIMES), CD(NODE_LATENCY), CD(NODE_BANDWIDTH),
          CD(FILE_SIZE), CD(JOB_LENGTH),
      CD(CPU_SPEED), CD(FILE_GENBY), CD(JOBFINISH), CD(KERNEL_STATUS)
      );
    errcheck();
    to_host(M(KERNEL_STATUS));
    node_not_finished = M(KERNEL_STATUS)->data[KS_NODE_NOT_FINISHED];
    job_improved = M(KERNEL_STATUS)->data[KS_JOB_IMPROVED];
    job_done = M(KERNEL_STATUS)->data[KS_JOB_DONE];
    int debug = M(KERNEL_STATUS)->data[KS_DEBUG];
    //printf("not-fin %d imp %d done %d debug %d\n", node_not_finished,
        job_improved, job_done, debug);

    /*to_host(M(NODELASTJOBFINISH));
    printf("last job finish\n");
```

```
    mtx_print(M(NODELASTJOBFINISH), stdout);
    printf("max %d\n", mtx_max_gpu(M(NODELASTJOBFINISH)));*/

    /*to_host(M(NODECOMPLETEDJOBS));
    printf("node completed jobs\n");
    mtx_print(M(NODECOMPLETEDJOBS), stdout);

    to_host(M(NODEJOBCOUNT));
    printf("node job count\n");
    mtx_print(M(NODEJOBCOUNT), stdout);*/

    //to_host(M(NODECOMPLETEDJOBS)); mtx_print(M(NODECOMPLETEDJOBS),
        stdout);

    //to_host(M(NODEJOBCOUNT)); mtx_print(M(NODEJOBCOUNT), stdout);
    //usleep(100000);
  }
  //printf("\n");
  //printf("max %d\n", mtx_max_gpu(M(NODEJOBCOUNT)));

  int tmax = mtx_max_gpu(M(NODELASTJOBFINISH));

  // to_host(M(NODELASTJOBFINISH)); printf("gpu lastjobfinish:\n");
      mtx_print(M(NODELASTJOBFINISH), stdout); printf("max=%d\n", tmax);

  to_host(M(JOBFINISH));
  //mtx_print(M(JOBFINISH), stdout);

  return tmax;
}

void opt_sim_gpu_a()
{
  //mtx_print(M(NODELASTJOBFINISH), stdout);
}
```

Listing 8: Makefile

```
CC=gcc
CFLAGS=-Wall -Werror -DMEXP=19937 -Wno-unused-but-set-variable
    -Wno-unused
DEPS = sfmt/SFMT.h matrix.h

.PHONY : all

all: genfiles main_cpu

%.o: %.c $(DEPS)
  $(CC) -c -o $@ $< $(CFLAGS)

genfiles: genfiles.o sfmt/SFMT.o matrix.o
  gcc -o $@ $^ $(CFLAGS)

cuda.o: cuda.cu
  nvcc -arch sm_12 -c -o $@ $<
```

```
main_cuda.o: main.c
  $(CC) -c -o main_cuda.o main.c $(CFLAG) -DCUDA

#$(CFLAGS)

cudatest: cuda_test.o cuda.o matrix.o sfmt/SFMT.o
  nvcc -o $@ $^

#--compiler-options $(CFLAGS)

main_cuda : cuda.o main_cuda.o sfmt/SFMT.o matrix.o
  nvcc -o $@ $^

main_cpu: main.o matrix.o sfmt/SFMT.o
  gcc -o $@ $^ $(CFLAGS) -lrt -lm
```

kod

Univerzita Mateja Bela v Banskej Bystrici

Fakulta prírodných vied

Optimalizácia plánovania úloh v gridoch a cloudoch pomocou GPGPU

diplomová práca – systémová príručka

Príloha B

Banská Bystrica, 2014 Bc. Michal Povinský

Obsah

1 Funkcia programu

Úlohou programu je optimalizácia rozvrhu úloh na gride pomocou grafickej karty. Najzložitejšou časťou programu je simulátor vykonávania úloh, ktorý sa používa na vyhodnocovanie plánu úloh podľa času dokončenia úloh. Na hľadanie optimálneho riešenia slúži optimalizátor, ktorý hľadá optimálny rozvrh, alebo rozvrh približujúci sa k optimu.

Program k svojej činnosti nevyžaduje žiadne databázy.

2 Analýza riešenia

Proces simulácie pozostáva z týchto krokov:

1. inicializácia – príprava dátových štruktúr
2. vyhodnotenie času prenosu súborov – výpočet času súborov
3. výpočet času spracovania úloh – výpočet času potrebného na spracovanie jednotlivých úloh
4. opakovanie až kým sa výsledok prestane meniť

Proces optimalizácie pozostáva z týchto krokov:

1. inicializácia – vygenerovanie náhodného rozvrhu
2. vyhodnotenie – náhodná zmena rozvrhu
3. vyhodnotenie nového rozvrhu
4. porovnanie a rozhodnutie o akceptovaní
5. kontrola ukončovacích podmienok a opakovanie pri nesplnení ukončovacích podmienok

Rozdiel v implementácii programu na CPU a GPU je v simulácii, ktorá sa v prípade implementácie na GPU vykonáva na grafickej karte.

3 Popis programu

Simulátor pracuje podľa algoritmu zobrazeného na obrázku č. 1 a optimalizátor podľa algoritmu na obr. č. 2.

Ako optimalizačný algoritmus bolo použité simulované žíhanie.

Parametre programu (počet iterácií, prípadne ďalšie) sa nastavujú v zdrojovom kóde. Predkladaná aplikácia nemá implementované grafické rozhranie, spúšťa sa pomocou príkazového riadku. Tiež výstup smerovaný na obrazovku je v príkazovom riadku. Po štarte program načíta vstupné dáta a spustí optimalizáciu. Počas optimalizácie zapisuje priebežné výsledky optimalizácie do súboru. Program skončí, keď vykoná nastavený počet iterácií.

Pre vykonávanie funkcie programu je potrebný program generujúci náhodné vstupy – generátor náhodných čísel.

3.1 Popis vstupných a výstupných súborov

Program očakáva ako jediný parameter názov vstupného súboru. Vstupný súbor sa skladá z niekoľkých matíc. Matice sú zapísané v tvare

```
nazov_matice
matrix stlpce riadky
1 2 3 ...
...
```

Obsahom matíc je:

- `node_latency` – oneskorenie komunikácie medzi uzlami
- `cpu_speed` – rýchlosť procesorov jednotlivých uzlov
- `job_length` – dĺžka vykonávania úloh
- `file_size` – veľkosť súborov
- `file_genby` – číslo úlohy, ktorá generuje súbor
- `job_files` – závislosti úloh na súboroch
- `file_mirrors` – umiestnenie súborov
- `node_bandwidth` – rýchlosť prenosu medzi uzlami

Výstupný súbor sa skladá z jedného riadku pre každú iteráciu. Riadok obsahuje nasledujúce hodnoty:

- čas dokončenia poslednej úlohy
- čas dokončenia poslednej úlohy pre prehľadávaný rozvrh
- čas dokončenia všetkých úloh

- čas dokončenia všetkých úloh pre prehľadávaný rozvrh
- hodnota kriteriálnej funkcie
- hodnota kriteriálnej funkcie pre vyhodnocovaný rozvrh
- teplota

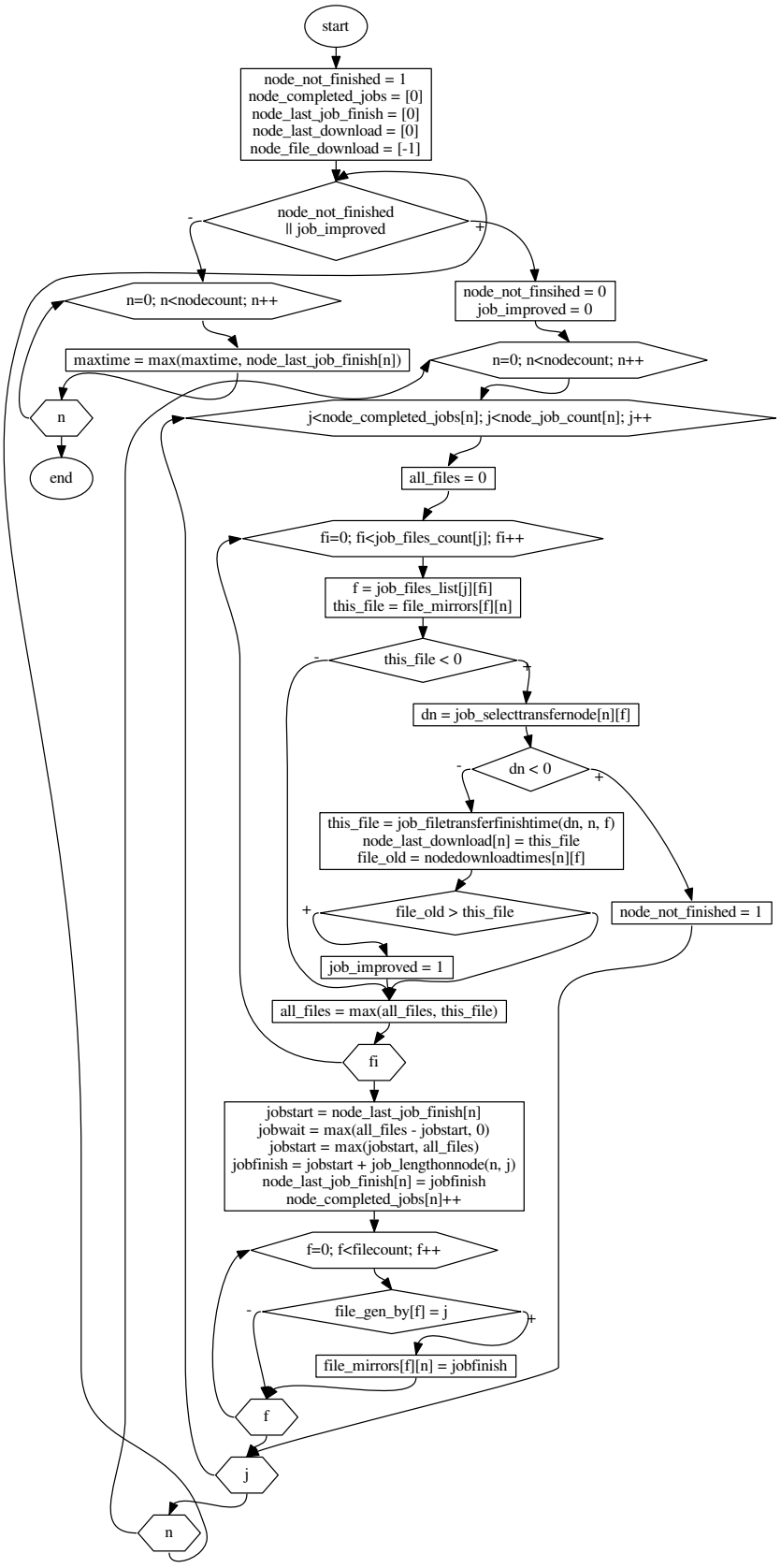
4 Kompilácia programu

Zdrojové kódy sú napísané v jazyku C. Kompiláciu je možné vykonať príkazom `make`. Program pre GPU je možné skompilovať príkazom make `main_cuda`.

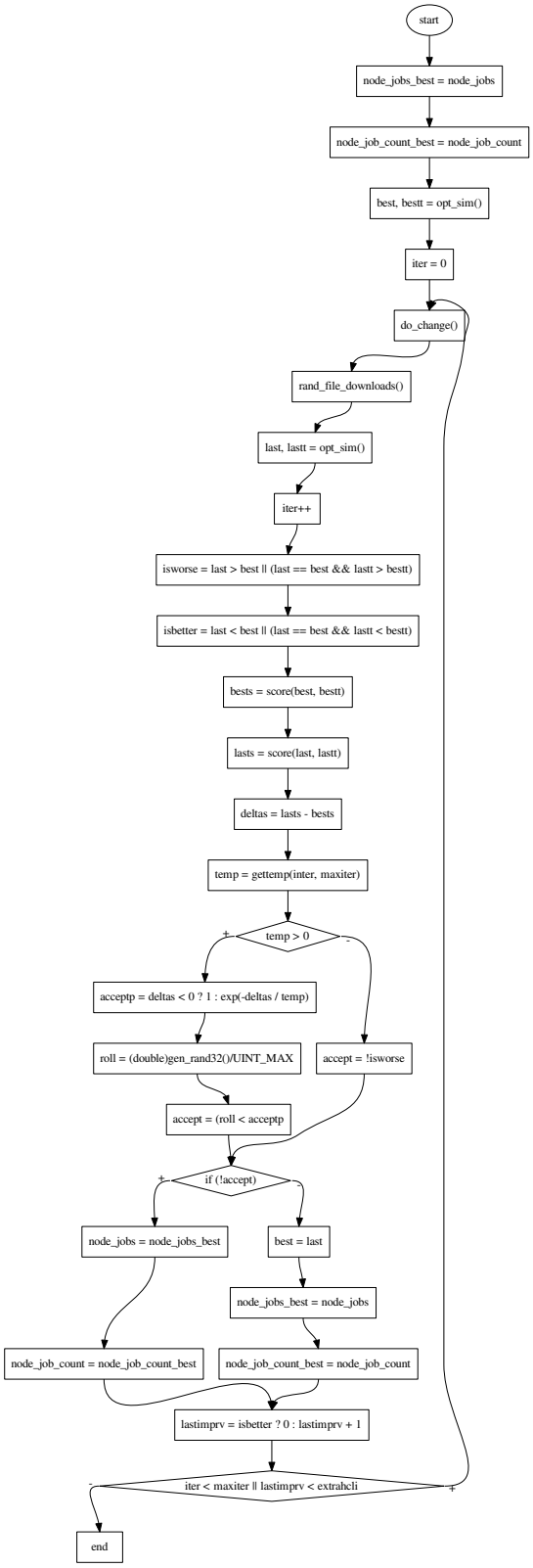
5 Požiadavky na technické prostriedky

Pri vývoji programu bola použitá nasledovná počítačová zostava:

- procesor Intel I7-720QM
- 8Gb RAM
- grafická karta NVIDIA Geforce GTS360M



Obr. 1: Simulátor



Obr. 2: Optimalizačný algoritmus

Univerzita Mateja Bela v Banskej Bystrici

Fakulta prírodných vied

Optimalizácia plánovania úloh v gridoch a cloudoch pomocou GPGPU

diplomová práca – používateľská príručka

Príloha C

Banská Bystrica, 2014 Bc. Michal Povinský

Obsah

1 Funkcia programu

Programový systém predstavuje implementáciu algoritmu simulovaného žíhania pre optimalizáciu plánovania úloh v gride pomocou GPGPU. Jedná sa o NP-úplný problém neriešiteľný bežnými algoritmami. Vyvinutý algoritmus vyhľadáva riešenia približujúce sa k optimu. Teoretický základ riešenia je popísaný v hlavnej časti diplomovej práce.

2 Požiadavky na technické prostriedky

Program nemá žiadne špeciálne požiadavky v prípade behu na CPU, ale v prípade behu na GPU potrebuje funkčné prostredie CUDA. Bol otestovaný na počítači s procesorom Intel7-720QM, RAM 8 Gb a grafickou kartou NVIDIA Geforce GTS 360m.

Program bol vyvíjaný a testovaný na operačnom systéme Debian GNU/Linux.

3 Požiadavky na programové prostriedky

Aby bol program funkčný je potrebné mať nainštalované prostredie NVIDIA CUDA.

4 Technické pozadie

Systém bol programovaný v jazyku C.

5 Použitie programu

Program sa spúšťa príkazom `./main vstupny_subor`, prípadne `./main_cuda` pre GPU verziu. Po spustení prebehne optimalizácia a výstup sa uloží do súboru `main.log`. Optimalizácia je ukončená po prebehnutí vopred nastaveného počtu iterácií.

6 Popis pracovných súborov

Program pracuje s dvoma súbormi: vstupným a výstupným. Vstupný súbor obsahuje informácie o vlastnostiach gridu a úloh. Výstupný obsahuje informácie o vývoji času dokončenia úloh.

7 Obmedzenie programu

Priame obmedzenia programu nie sú známe. Závisia na veľkosti pamäte na grafickej karte.

8 Zhodnotenie riešenia

Tento program predstavuje implementáciu navrhnutého algoritmu na procesore a na grafickej karte.

Riešenie je vhodné na experimentálne overenie zrýchlenia výpočtov na GPU oproti CPU. Reálne využitie systému si vyžaduje prispôsobenie reálnemu gridovému prostrediu.