IB109 poznamky – 2020

01 - Uvod

Subeznost – existencia dvoch alebo viacerych procesov sucasne

Vypocetny paralelizmus

- Na vsetkych urovniach, od registrov az po koexistenciu rozsiahlch distribuovanych aplikacii
- Chceme ho kvoli vyssiemu vykonu
- Zaroven je nutny kvoli priestorovej distribucii
- To vsak za cenu zlozitejsieho navrhu a ceny

Sekvencne algoritmy, slozitostna trieda NC

Dovody pre vyvoj paral. App.:

- Vykonnost – efektivne vyuzitie aggregovane vyp. Prostredieky -> rychlejsi vypocet
- Proveditelnost – v pripade kedy agregacia vyp. Sily nie je volba, ale nutnost -> aby sa vypocet skoncil v rozumnom case, pretoze kvoli narocnosti by inac trval pridlho
- Bezpecnost – duplikacie casti system, ak by niektora havarovala alebo bola narusena jej dovera
- Cena – agreg. Vyp sila je lacnejsia a zaroven je lacnejsie udrziavat nesuviciace casti aplikacie

Abstraktny model vypocetneho system:

Procesor – Datova cesta – Datove ulozisko

Vsetky casti system mozu byt uzkym miestom voci vykonosti aplikacie

Paralelizmus je prirodzeny sposob prekonania uzkych miest

Procesory

Neustala nutnost zvysovat vykon, pricom vykon je spajany s **Moorovym zakonom**

Moorov zakon

- pomenovany podla spoluzakladatela Intelu, Gordona Moore
- Hovori ze pocet tranzistorov sa v procesoroch zdvojnasobi cca kazdych 18 mesiacov -> NEDAL sa uplne dodrzat v jednom jadre, zacalo sa pre zvysenie vykonu zavadzat viacero jadier, viz nissie

Metody zvysovania vykonu procesoru

- Zvysovanie frekvencie vnutornych hodin
- MULTIPLICITA, PARALELIZMUS

Pre pekne vyobrazenie moorovho zakona a poctu tranzistorov, vygoogli si “transistor counts and moore law”

Trendy vo vyvoji procesorov

- Nedari sa zvysovat vykon jedneho jadra tak ako by sa chcelo
- Fyzikalne zakony brania neustalej miniaturizacii – okolo 5 nm sa nedaju udrzat elektrony v atome
- Sucasne je to 14-16nm

Riesenie

- Viacero jadier procesoru
- Pravdepodobne takyto vyvoj bude pokracovat aj dobuducna
- Je to odklon od monolityckych jadier k desiatkam mensich specializovanym ci hybridnym rieseniam

Dosledok

- Sekvencne algor. Nemozu nadalej tazit z rastuceho vykonu cpu
- Inu PARALELIZMUS VYPOCTU JE NEVYHNUTNY SMER VYVOJA

Datove cesty – paralelizmus v komunikacii

- Vetsia priepustnost komunikacnych liniek s nasledkom nizsej latencie
- Robustnost a spolahlivost komunikacnych liniek
- Napriklad sirka zbernice 32/64/128 bitov a dualband wifi router

Pamet

- Vykon procesorov prevysuje vykon ostatnych component
- To znamena ze cesta processor – pamet – disk je pridlha
- Doba pre ziskanie jednotky informacie z pameti rastie s vzdialenostou miesta ulozenia od miesta zpracovania

Viacurovnove ulozenie informacii

Registry procesoru

L1/L2/L3 cache

Operacna pamet

Cache I/O zariadeni

Magneticke/opticke disky

Cache

- Kopia casti dat v rychlo dostupnom mieste
- Moze a nemusi byt kontrolovana userom alebo OS

Priklad roznych moznosti kontroly

- L1/L2 cache v ramci CPU nemozu byt kontrolovane programatorom
- I/O efektivne algoritmy
    - Obchadzaju virtualizaciu pameti kontrolovanou OS a miesto toho realizuju vlastny sposob pouzitia operacnej pamete ako cache pre data na disku

Multiplicita pametovych modulov

- Vacsie mnozstvo ulozitelnych dat
- Vacsie mnozstvo liniek do pameti, tj, vyssia priepustnost
- Vacsia rezie na udrzanie konzistencie
- Napr diskove pole, P2P siete, NUMA architetury
    - NUMA – viac procesorove pocitace s viacerimi pametovymi modulmi usporiadanymi tak, ze pristup jedneho procesoru do roznych pametov modulov je rozne rychly. Obvykle je pamet blizko k urcitemu CPU, ale architektura umoznuje citanie danej pameti inym cpu

Paralelne vypocty

Paralelizmus z pohladu OS

Multitasking na jednom jadre

- Falosny multitasking, aplikacie sa na CPU striedaju pocas behu
- Zdanlive “bezi” viacero aplikacii
- Jednotka planovania OS je process

Multitasking na viacerych vypocetnych jadrach

- Rozne aplikacie maju priradaene rozne vyp. Jadra
- Inak standardny multitasking na kazdom jednom jadre
- Jednotka planovanie OS je process

Multitasking a multithreading

- Kazda aplikacia moze mat viac vypoc. Vlakien
- Vlakna sa v behu na jednom CPU jadre striedaju
- Vlakna jednej aplikacie mozu bezat na roznych jadrach
- Jednotka planovania OS je vlakno

Flynova klasifikacia

Single Instruction Single Data - SISD

- V dany okamzik je zpracovana len jedna instrukcia and jednym datovym prudom
- Klasicky sekvencny vypocet

Single Instruction Multiple Data - SIMD

- Jedna istrukcia je vykonavana and viacerymi datovymi prudmi
- Velktorove instrukcie CPU, arhcitektura GPU

Multiple Instruction Multiple Data - MIMD

- Nezavisly sucasny beh dvoch a viacerych SISD, SIMD pristupu

Multiple Instruction Single Data – MISD

- Prakticky sa nevyskutuje

Distribuovany/ paralelny system

- Specifikovany po castiach, tj procesoch
- Chovanie system vznika interakciu sucasne beziacich procesov
- Emergentne javy, tj, chovanie ktore neni nutne explicitne naproramovane/ mienene, ale nastane zhodou okolnosti

Synchronizacia – omedzenie na prekladanie a subeh akcii jednotlivych procesov distribuovaneho system

Komunikacia – prenos informacie z jedneho procesu na druhy

Vybrane problem distrib. Systemov

Javy

- Nekonzistentna vizia konzistentneho sveta
- Vzajomna interferecia

Rizika

- Race-condition, tj pokus o zmenu dat dvoma procesmi naraz, vysledne data zavisia od poradia vykonania procesov - nejasne
- nederministicke chovanie
- Uviaznutie(deadlock,livelock)
- Starnutie, hladovenie(starving)
- Pretecenie buferov, problem producent-konzument(producent nemoze produkovat ak je buffer konzumenta plny, konzument nemoze konzumovat ak je buffer prazdny)
- Zbytocna strata vykonu aktivnym cakanim

Efektivny algoritmus s snazi sam zistit aka pamet je vhodna, napriklad database.

8 CPU, 9 vlakien -> 1 vlakno sa prehadzuje medzi 8 cpu, ostatnych 8 vlakien pevne pridelenych, pretoze ak by boli zvlast 2 vlakna na jednom CPU, tak by bolo dane CPU 2x pomalsie

Dovody vyssej narocnosti vyvoja paralelnych systemov

- Nutnost specifikace souběžných úkolů.
- Nutnost specifikace koordinace úkolů.
- Paralelní algoritmy.
- Nedostačující vývojová prostředí.
- Nedeterminismus při simulaci paralelních aplikací.
- Absence reálného modelu paralelního počítače.
- Rychlý vývoj a zastarávání použitých technologií.
- Výkon aplikace náchylný na změny v konfiguraci systémů

02 – programovanie v prostredi so zdielanou pametou

HW platformy

Paralelne systemy so zdielanou pametou

- Systemy s viac procesormi
- Systemy s viac jadernymi procesormi
- Systemy s procesormi so zabudovanym SMT – Simultanny multithreading
- Kombinacia

Rizika paralelnych vypoctov na sucasnych procesoroch

- Mnoge optimalizacie na urovni procesorov boli navrhnute tak, aby zachovaly semantiku sekvencnych programov

POZOR na

- Preusporiadanie instrukcii
- Odlozenie zapisu do pameti

SMT – Simultanny multithreading

Princip

- Procesor vyuziva prazdne cykly sposobene latenciou pameti k vykovananiu instrukcii ineho vlakna
- Vyzaduje duplikaciu urcitych casti procesoru, napr. Registrov
- Vlakna zdielaju cache
- Prepnutie vlakna urobi kopiu dat

Priklad: Intel Pentium 4

- Hyper-Threading Technologgy – **HTT**
- OS s podporou SMP(symetricky multiprocessing) vidi system s SMT/HTT ako viac jadrovy system
- Az 30% narast vykonu, ale vzhladom k sdielanej cache moze byt rychlost vypoctu jedneho vlakna nizsia

Viac-jaderne procesory – multicore

Viac plnohodnotych procesorov v jednom chipe

Vyhody

- Efektivnejsia cache koherencia na najnizsej urovni
- Nizsie naklade pre koncoveho usera

Nevyhody

- Viac jadier emituje vacsie zbytkove teplo(globalne oteplovanie)
- Takt jedneho jadra byva nizsi
- Automaticke docasne podtaktovanie/pretaktovanie
- Jadra zdielaju datovu cestu do pameti – pravdepodobne uzka cast, teda bottleneck

Realita

- Viac jadrove procesory so SMT
- Intel Core i7 – hexa core so SMT ma az 12 paralelnych jednotiek

Paralelizmus v prostredi so sdielanou pametou

Idealizovany model

- Tu sa riesi navrh paralelneho algoritmu
- Jednotlive vypocetne jadra paralelneho system pracuju uplne nezavisle
- Pristup k datam pameti je bezcasovy a navzajom sa jednotlive pristupy vylucuju

- Komunikacia uloh prebieha atomicky cez zdielane datovy struktury

Realita

- Na tejto urovni programator riesi technicku implementaciu paral. Alg.
- Pristupy do pameti cez zbernicu su prilis pomale pre CPU
- Registry procesorov a cache pameti – rychle kope maleho mnozstva dat na roznych miestach datovej cesty
- Problem koherencie dat – data rovnake by mali byt rovnake pre vsetky procesy

Paralelne ulohy v kontexte OS – procesy a vlakna

Procesy

- Skryvaju pred ostatnymi procesmi svje vypocetne prostriedky
- Pre riesenie paralelnej ulohy je potreba medzi procesova komunikacia – **IPC**
  Zdielane pametovy segment, sokety, pomenovany a nepomenovane rury

Vlakna

- Existuju v kontexte jedneho procesu
- V ramci rodicovskeho procesu zdielaju vypocetne prostriedky
- Komunikacia prebieha cez zdielane datove struktury
- Ucelom interakcie je skor synchronizacia nez transport dat
- Subjekty procedury planovania

Priklad pouzitia vlakien vramci procesu

*****

Vlakno

- Realizuje vypocet, tj sekvenciu instrukcii
- Kazd process je tvoreny aspon jednym vlaknom
- Hlavne vlakno procesu vytvara dalsie vlakna

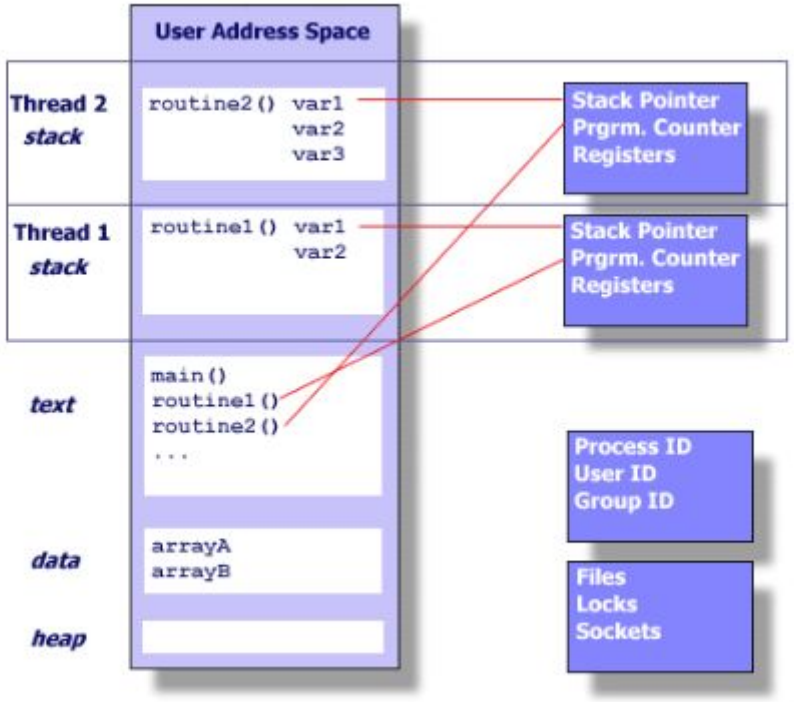
Proces ma

- Itentifikatory procesu a vlastnika
- Premenne prostredia, pracovny adresar
- Kod
- Registry, zasobnik, halda
- Odkazy na otvorene suboy a zdielane knihovne
- Reakcie na signaly
- Kanally IPC

Vlakna maju private

- Zasobnik

- Registry
- Frontu signalov



Argumenty pro pouzivanie vlakien miesto procesov

Vykon aplikace

- Vytvorenie procesu je vyrazne drahsie nez vytvorenie vlakna
- Zmena dat prevedena vramci jedneho vlakna je viditelna v kotnexte celeho procesu
- Komunikacia medzi vlaknami spociva v predavani referencie na data, nie predavaniu obsahu
- Predavanie referencie sa deje vamci jedneho procesu, operacny system nemusi riesit skryvanie dat a pristupove prava

Nevyhody

- Vlakna nemaju ziadne “sukromie”
- Zdielane globalne premenne

Efektivne vyuzite cache

Princip cache

- Mensia, ale rychlejsia pamet na pomalej datovej ceste
- Pri prvom nacitani sa okolie citanej informacie ulozi do cache
- Pri naslednom citani z okolia povodnej informacie staci citat z cache

Koherence

- Soulad dat ulozenych v pameti pocitaca, respective cache
- Je zajistene, ze existuje prave jedna platna hodnota asociovana s danym pametovym miestom
- Z dovodu rychlosti su zapisy procesoru do pameti odkladane a zdruzovane, blizsie specifikacie chovanie procesoru v tomto ohlade je dana **pametovym modelom** daneho CPU

Suvisiace pojmy

- Cache line – atomicky pametovy blok ulozeny v cache
- “hit ratio” - cislo vyjadrujuce uspesnost obsluzenia poziadavkov na data datami ulozenymi v cache
- “vyliatie cache” – procedura aktualizacie dat v pameti hodnotami ulozenymi v cache

Zasady efektivneho pouzitia cache

- Casova lokalita – pristupy v malom casovom intervale
- Priestorova lokalita – pristup k datom ulozenym adresne blizko seba
- Zarovnana alokacia pameti – napr. Memalign v GNU C

False sharing

- Paralelny cache koherentny system s viac procesormi
- Program s niekolkonasobne viacero vlaknami
- Pole hodnot int pole[nr_of_threads]
- Vlakna pocitaju vyslednu hodnotu typu int a k vypoctu si ukladaju medzivysledok typu int

Varianty implementacie

A) Kazde vlakno opakovane zapisuje do datoveho pola integerov na poziciu urcenu jeho ID
B) Kazde vlakno zapisuje do lokalnej premennej a pred skoncenim nakopiruje hodnotu do pol ana poziciu urcenu jeho ID

*****

Pouziva sa cache line filler na zvysenie vykonu.

Pricom nie je garantovane ze vsetky informacie ulozene do explicitne definovanych premennych budu zapisane do pameti a zaroven premenne mozu byt realizovane registrom procesoru.

A ako dosledok, postupnost hodnot ulzoenych do jednej premennej vramci jedneho vlakna moze byt videna **ciastocne** alebo **vobec.**

Udrzovanie koherencie cache pameti je naklade a zaroven sucasne prekladace **nevynucuju** aby kazda vypocitana hodnota bola ulozena do pameti.

Ako dosledok, modifikacie v jednom vlakne CPU, sa nemusi prejavit v inom vlakne na inom CPU.

Problem:

- Vlakno P0 neskonci alebo nezaznamena zmenu premennej x
- Moze zavisiet aj na stupni optimizacie prekladacu
    - Chyba sa neprejavi pri g++ -00
    - Ale prejavi sa pri g++ -02

Riesenie

- Je nutne oznacit premenu ako nestalu premenu, tj, **volatile**(c, c++)
- Prekladac zaisti aby dana premenna nebula realizovana iba na urovni registrov CPU, ale pred a po kazdom pouziti bola nacitana/ulozena do pamete

Rozlisujeme rozdielne typy volatile:

- Umiestnenie pred alebo za datovy typ v definicii premenej
- Rozlisujeme:
    - Ukazatel na nestalu premennu
        - Volatile T *a
    - Nestaly ukazatel na nestalu premennu
        - Volatile T * volatile a
    - Nestalu premennu
        - Volatile T a
        - T volatile a

Pripady kedy je nutne pouzit volatile:

- Premenna zdielana medzi subeznymi vlaknami/procesmi
- Premenna zastupujuca vstupny/vystupny port
- Premenna modifikovana procedurou obsluhujucu prerusenie

Presny popis toho ako processor manipuluje so zapismy dat do pameti je sucastou specifikacie procesoru, jedna sa o tzv pametovi model.

*Napriklad, zapisy hodnot do pamete niesu vykonane v okamihu zpracovania instrukcie, ale su odkladane na neskor a zhlukovane. Preto napriklad pocitanie do 200000 na systeme s jednou vykonovou jednotkou je vzdy 200000, ale na paralelnych architekturach je castokrat mensie.

Pozor:

- Poradie zapisov do pameti podla instrukcii CPU sa nemusi zhodovat so skutocnym poradim zapisu hodnot do pameti
- Pametovy model garantuje korektnost **iba** pre sekvencne programy

*Paralelne programovanie si vyzaduje opony bod na urovni HW, inac takmer nemozne.

Pametova bariera

- HW primitivum pre synchronizaciu stavu pamete a stavu procesorov v danom mieste programu
- Na sucasnych procesoroch realizivoane instrukciou **mfence**
- Na hardwaovej urovni prevedie serializaciu vsetkych *load* a *store* instrukcii, ktore sa vyskytuju pred instrukciou *mfence*. Tato serializacia zaisti ze efekt vsetkych instrukcii pred instukciou *mfence* bude globalne viditelny pre vsetky instrukcie nasledujucich za instrukciou mfence

HW podpora pre atomicke instrukcie

- Pametova bariera neriesi problem atomickych instrukcii ako su napriklad TEST-AND-SET, COMPARE-AND-SWAP
- Instrukcie spomenuteho typu su vsak pre ucely efektivneho paral. Progr. Velmi vhodne

Dalsia HW podpora

- Alpha, Mips, PowerPC, ARM: instrukcie typu LL(Load Linked)/SC(Store Conditional)
    - LL vrati sucasnu hodnotu pamete na adrese, pricom nasledujuci SC ulozi na rovnaku adresu nove data, iba ak nedoslo k zmene na adrese od casu vykonania LL
- X86 architektura
    - Lock – nasledujuca instrukcia ktora zapisuje do pamete prebehne atomicky, a jej efekt bude hned globalne viditelny
    - XCHG – prehodi obsah registru a pametoveho miesta -> obsahuje z definicie prefix *lock*

Moznosti realizacie atomcikych instrukcii na urovni kodu

1. Jazyk symbolickych adries – JSA -> fancy pojem pre assembler
2. Zabudovane funkcie prekladaca, od GCC 4.1
    a. `type __sync_val_compare_and_swap`
    b. `type __sync_fetch_and_add`
3. sucast programovacieho jazyka – napriklad C++ rev 11, Java, …

*

03 – programovani v prostredi so zdielanou pametou vol 2.

Rizika spojene so zdielanou pametou

Paralelne programy mozu byt opakovanom spusteni zdanlive nahodne vykazovat rozne chovania

Vysledok programu moze zavistiet na absolutnom poradi prevedenia instrukcii programu, tj prelozenie instrukcii zucastnenych procesov/vlakien

Race condition

- nedokonalsot paralelneho programu, ktora sa prejavuje hore spominanym sposobom sa oznacuje **race condition,** zkratene **race**

Atomicita operacii

- jednoduchy prikaz vo vyssom programovacom jazyku neodpoveda nutne jednej instrukcii procesoru
- v modernych OS je kazde vlakno podrobeene planovaciemu procesu
- vykonanie postupnosti instrukcii procesoru odpovedajucemu jednemu prikazu vyssie program. Jazyka moze byt prerusene a prelozene vykonanim instrukcie ineho vlakna
- napriklad, priradenie cisla do premennej efektivne moze znamenat nacitanei do registru, prevedenie aritmetickes operacie a ulozenie vysledku do pameti
    - * moze sa stat ze sa efekt jedneho priradenia do zdielanej globalnej premennej zcela strati

Relativna rychlost vypoctu *

Neda sa spoliehat na sucasny subeh vlakien a teda aj na relativnu rychlost vypoctu jednotlivych vlakien

Uviaznutie – Deadlock

- Pokial maju vlakna **neusporiadane inkrementalne** poziadavky na **unikatne** zdielane zdroje, moze dojst k deadlocku

Ak uviaznu 2 vlakna z viacerych, jedna sa stale o uviaznutie.

Ciastocny deadlock sa velmi narocne detekuje

Hladovenie, Starnutie, nepogrese – Livelock *

Jedna sa o jav, kedy aspon jedno vlakno nie je schopne k vzhladom k paralelnemu subehu s inym vlaknom pokrocit vo vypocte za danu hranicu

Thread-safe procedura

Oznacuje procedure ci program, ktoreho kod je bezpecne priviest(vzhladom k semantike vystupu a stabilite vypoctu) subezne s niekolkymi vlaknami bez nutnosti vzajonem synchronizacie

Ak sa k datovym strukturam pristupuje naraz z viacerych vlakien, nejedna sa o thread-safe pripad

Knihovnove funkcie nemusia byt thread-safe

- Rand() -> rand_r()

Re-entrantna procedura

Jedna sa o procedure, ktorej prevedenie mzoe byt vramci jedneho vlakna prerusene, kod kompletne vykonany od zaciatku do konca v ramci tej istej ulohy, a potom znovu obnoveny/dokonceny po preruseni kodu

Je to termin z cias ked este neboli multitaskingove OS

Re-entrantna procedura nemusi byt thread-safe a rovnako thread-safe procedura nemusi byt re-entrantna *

Nebezpecne akcie vzhladom k paralelnemu zpracovaniu

- Nekontrolovany pristup k global. Premen. A halde
- Uchovavani stavu procedry do globalnych premennych
- Alokacie a dealokacie zdrojov globalneho rozsahu(subory,…)
- Nepriamy pristup k datam skrz odkazy alebo ukazatele
- Viditelny vedlajsi efekt - > napr modifikacie nestalych premennych

Bezpecne strategie

- Pristup iba k lokalnym premennym(zasobnik)
- Kod je zavisly iba na argumentoch danej funkcie
- Akekolvek volane podprocedury a funkcie su thread-safe *

Pristup k zdielanym premennym

Pristupvoanie k sdielanym premennym je nekorektne

Vsetky modifikacie a neatomicke citania globalnych premennych musia byt **serializovane**

Kriticka sekcia

- Cast kodu ,ktorej prevedenie je neprelozitelne instrukciami ineho vlakna
- Realizacia kritickej sekcie musi byt odolna voci planovaniu

Zamykanie

- Vlakno vstupujuce do volnej kritickej sekcie svojim vstupom znemozni pristup ostatnym vlaknam
- Ostatne vlakna cakaju pred vstupom do kritickej sekcie
- Pri odchode z kritickej sekcie sa zamok uvolni a ziska ho **nahodne** cakajuce vlakno

Jednoduche riesenie zamku

- Zdielana atomicka pristupova bitova premenna, ktorej hodnota indikuje pritomnost procesu ci vlakna v priradenej kritickej sekcii
- Premenna je manipulovana pri vstupe a vystupe z/do kritickej sekcie
- Vyzaduje podporu HW pre atomicku manipulaciu

Aktivne cakanie – spinlock

Dokial neuspeje, vlakno sa bude snazit dostat do sekcie opakovane

Uspavanie

Procesy/vlakna sa po neuspechu vstupit do kritickej sekcie uspia

Zobudia sa po vyprisani casoveho limitu alebo explicitne inym vlaknom

Rizika zamykania

- Uviaznutie, starnutie, znizenie vykonu

Manipulacia so zamkom vynucuje vylatie cache pameti

Mnoho pristupov k zamykanym premennym moze byt uzkym miestom vykonu aplikacie, z principusa ale neda odstranit

Petersonov algoritmus – spinlock, user-space

- Spravodlivy algoritmus preriadenie vzajomneho vylucenia
- Nesposobuje starnutie ani uviaznutie
- Vyzaduje atomicke zapisy do premennych
- Citlivy na vykonavanie instrukcii mimo poradie

POSIX thread API

Zakladne delenie funkcionality

Sprava vlakien

- Vytvaranie, oddelovanie a spojovanie vlakien
- Funkcie na nastevenie a zistenie stavu vlakna

Vzajomne vylucenie – mutexes

- Vytvarenie, nicenie, zamykanie a odomykanie mutexov
- Funkcie na nastavenie a zistenie atributov spojenych s mutexami

Podmienkove/podmienenne premenne – conditional variable

- Sluzi pre komunikaciu/synchronizaciu vlakien
- Funkcie na vytvaranie, nicenie, “cakanie na” a “signalizovanie pri” specifickej hodnote podmienkovej premennej
- Funkcie na nastavenie a zistenie atributov premennych

Posix standard

Cez 60 API funkcii

- #include <pthread.h>
- Preklad s volbou -pthread

Mnemotechnicke predpony funkcii

- pthread_, pthread_attr_
- pthread_mutex_,pthread_mutexattr_
- pthread_cond_, pthread_condattr_
- pthread_key_

Pracuje sa so skrytmi objektmi – opaque objects

- Objekty v pameti o ich podobe programator nic nevie
- Pristupovane k vyhradne pomocou odkazu (handle)
- Nedostupne objekty a neplatne(dangling) referencie

Atributy objektu

Idea

- Vlastnosti vsetkych vlakien, mutexov a podmienkovych premennych nastavovane specialnymi objektmi
- Niektore vlastnosti entity musia byt specifikovane uz v dobe vzniku entity

Typy atributovych objektov

- Vlakna: pthread_attr_t
- Mutexy: pthread_mutexattr_t
- Podmienkove premenne pthread_condattr_t

Vznik a destrukcia

- Funkcie _init a _destroy s odpovedajucou predponou
- Parameter odkaz na odpovedajuci atributovy object

Sprava vlakna

Vytvorenie vlakna

- Kazdy program ma jedno hlavne vlakno
- Dalsie vlakna musia byt explicitne vytvorene programom
- Kazde vlakno (I vytvorene) moze dalej vytvarat dalsie vlakna
- Vlakno je vytvorene funkciou pthread_create
- Vytvorene vlakno je hned pripravene k pouzitiu
- Moze byt planovacom spustene skor nez sa dokonci volanie vytvaracej funkcie
- Vsetky data potrebne pri spusteni vlakna musia byt pripravene pred volanim vytvaracej funkcie
- Maximalny pocet vlakien je zavisly na implementacii

*

Pthread_create argumenty

thread_handle je odkaz na vytvorene vlakno

attribute odkaz na atributy vytvoreneho vlakna(NULL pre prednastavene nastavenie atributov)

thread_function ukazatel na funkciu noveho vlakna

arg ukazatel na parametry funkcie thread_function

pri uspesnom vytvoreni vlakna vracia 0

Ukoncenie vlakna nastava

- Volanim funkcie pthread_exit
- Pokial skonci hlavna funkcia rodicovskeho vlakna inak nez volanim pthread_exit
- Ak je zrusene inym vlaknom pomocou pthread_cancel
- Rodicovsky process je ukonceny (nasilne alebo volanim exit)

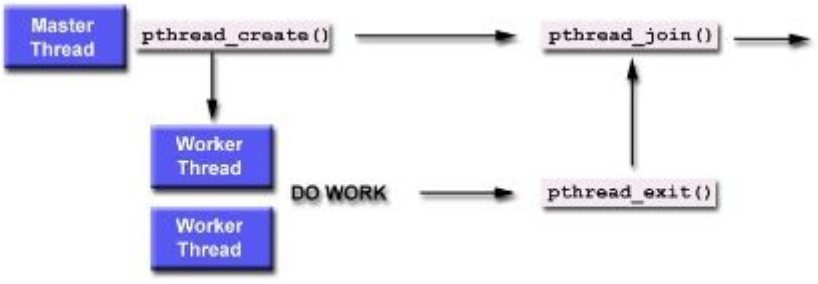
Void pthread_exit (void * value)

- Ukoncuje beh vlakna
- Odkazy na prostredie procesu (subory, IPC, mutexy) otvorene vramci vlakna sa nezatvaraju
- Data patriacie vlaknu musia byt uvolnene pred ukoncenim vlakna(system prevedie uvolnenie prostriedkov az po skonceni rodicovskeho procesu)
- Ukazatel *value* predany pri spojeni vlakien

*

Int pthread_join (pthread_t thread_handle, void **ptr_value);

- Caka na dokoncenie vlakna *thread_handle*
- Hodnota *ptr_value* je ukazatel na pointer specifikovany vlaknom *thread_handle* pri volani *pthread_exit*



-
- Nutny napriklad pokial main ma vraciat zmysluplnu navratovu hodnotu

Vlastnosti vlakien

Nespojitelne vlakna – detached threads

- Nemozu byt spojene volanim funkcie pthread_join
- Setria systemove prostriedky
- Prednastavene nastavenie typu vlakna nie je vzdy zrejme, preto je doporucene typ vlakna explicitne nastavit

*

Velkost zasobniku

- Minimalna velkost zasobniku nie je urcena
- Pri velkom pocte vlakien sa casto stave ze vyhradene miesto pre zasobnik je vycerpane
- Posix umoznuje zistit a nastavit poziciu a velkost miesta vyhradeneho pre zasobnik jedneho vlakna

*

Zrusenie vlakna

Int pthread_cancel (pthread_t *thread_handle);

- Vysle ziadost o zusenie vlakna *thread_handle*
- Adresovane vlakno sa moze a nemusi ukoncit
- Vlakno moze ukoncit sameho seba
- Pri zruseni sa prevedie vycistenie dat spojenych s rusenym vlaknom
- Funkcia skonci po odeslani ziadosti (je noblokujuca)
- Navratovy kod 0 znaci, ze adresove vlakno existuje, nie ze bolo/bude zrusene

Funkcia *pthead_cleanup_push* registruje cistiaci kod

Pomocou *pthread_setcancelstate* sa da nastavit ci je vlakno zrusitelne a ako

Vzajomne vylucenie

Motivacia

- Viacero vlakien vykona nasledujuci kod
  If (my_cost < best_cost) best_cost = my_cost;
- Nedeterministicky vysledok pre 2 vlakna a hodnoty:
  Best_cost==100, mycost@1==50, my_cost@2==75

Riesenie

- Umiestnenie kodu do kritickej sekcie
- Pthread_mutex_t

Inicializacia mutexu

```
int pthread_mutex_init (pthread_mutex_t
*mutex_lock,pthread_mutexattr_t *attribute)
```

- Parameter *attribute* specifikuje vlastnost zamku
- NULL znamena defaultne nastavenie

```
int pthread_mutex_lock (pthread_mutex_t *mutex_lock)
```

- Volanie tejto funkcie zamyka *mutex_lock*
- Volanie je blokujuce, dokial sa nepodari mutex zamknut
- Zamknut mutex sa podari iba raz jednemu vlaknu
- Vlakno, ktoremu sa podari mutex zamknut je v kritickej sekcii
- Pri opusteni kritickej sekcie, je vlakno "povinne" mutex odomknut
- Az po odomknuti je mozne mutex znova zamknut
- Kod vykonany vramci kritickej sekcie je po odomknito mutexu globalne viditelny(pametova bariera)

```
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock)
```

- Odomyka *mutex_lock*

Pozorovanie

- Velke kriticke sekcie znizuju vykon aplikacie
- Prilis casus a travi v blokujucom volani funkcie *pthread_mutex_lock*

Riesenie:

```
int pthread_mutex_trylock (pthread_mutex_t *mutex_lock)
```

- Pokusi sa zamknut mutex
- Vpripade uspechu vrati 0
- Vpripade neuspechu vrati *EBUSY*
- Zmysluplne vyuzitie komplikuje navrh programu
- Impementacia byva rychlejsia nez *pthread_mutex_lock(*nemusi sa manipulovat s frontami cakajucich procesov)
- Ma zmysel aktivne cakat opakovanym volanim trylock

Vlastnosti(atributy) mutexov

Normalny mutex

- Iba jedno vlakno moze jedenkrat zamknut mutex
- Pokial sa vlakno, ktora ma zamknuty mutex pokusi znova zamknut rovnaky mutex, dojde k uvaznutiu

Rekurzivny mutex

- Dovoluje jednemu vlaknu zamknut mutex opakovane

- K mutexu je asociovany citac(pocitadlo) zamknuti
- Nenulovy citac znaci zamknuty mutex
- Pre odomknutie je nutne zavolat unlock tolkokrat kolko bolo volane lock

Normalny mutex s kontrolou chyby

- Chova sa ako normalny mutex, akurat pri pokuse o druhe zamknutie ohlasni chybu
- Pomalsi, typicky pouzivany docasne pocas vyvoja aplikacie, potom nahradeny normalnym mutexom

```
int pthread_mutexattr_settype_np (ptrhead_mutexattr_t
*attribute,int type)
```

- Nastavenie typu mutexu
- Typ urceny hodnotou premenej *type*
- Hodnota *type* moze byt:
    - PTHREAD_MUTEX_NORMAL_NP
    - PTHREAD_MUTEX_RECURSIVE_NP
    - PTHREAD_MUTEX_ERRORCHECK_NP

04 – POSIX THREAD CONT., WIN32 THREADS

Zakladne delenie:

Sprava vlakien

- Vytvaranie, oddelovanie a spojovanie vlakien
- Funkcie na nastavenie a zistenie stavu vlakna

Vzajomne vylucenia – mutexes

- Vytvaranie, nicenie, zamykanie a odomykanie mutexov
- Funkcie na nstavenie a zistenie atributov spojenych s mutexami

Podmienkove/podmienene premenne – conditional variable

- Sluzia pre komunikaciu/synchronizaciu vlakien
- Funkcie na vytvaranie, nicenie, “cakanie na” a “signalizovanie pri” specifickej hodnote podmienkovej premennej
- Funkcie na nastavenie a zistenie atributov premennych

Podmienkove premenne

Motivace I

- Casto na jednu kriticku sekciu caka viacero vlakien
- Aktivne cakanie – permanentna zataz cpu

- Uspavanie timeoutom – netrivialna rezia, omedzena frekvencia dotazovania sa na moznost vstupu do kritickej sekcie

Motivacia II

- Vlakno realizuje ucelenu, logicky oddelenu funkcionalitu
- Ta neni potrebna za celu dobu behu aplikacie
- Programatorom riadena docasna deaktivacia vlakna

Obene riesenia:

- Uspanie vlakna, pokial vlakno ma/musi cakat
- Vzbudenie vlakna v okamihu ked je mozne pokracovat

Realizacia v POSIX Threads

- Mechanizmus oznacovany ako *podmienkove premenne*
- Podmienkova premena vyzaduje pouziti mutxu
- Po ziskani mutexu sa vlakno moze docasne vzdat tohoto mutexu a uspat sa (vramci danej podmienkovej premennej)
- Prebudenie musi byt explicitne signalizovane inym vlaknom

```
int pthread_cond_init (ptrhead_cond_t
*cond,pthread_cond_condattr_t *attr)
```

- Inicializuje podmienkovu premennu
- Ak ma *attr* hodnotu NULL, pouzije sa defaultne chovanie

```
int pthread_cond_destroy (ptrhead_cond_t *cond)
```

- Nici nepouzivanu podm. Premennu a suvisiace datove struktury

```
int pthread_cond_wait (ptrhead_cond_t
*cond,pthread_mutex_t *mutex_lock)
```

- Uvolni mutex *mutex_lock* a zablokuje vlakno v spojeni s podmienkovou premennou *cond*
- Po navrate vlakno opet vlastni mutex *mutex_lock*
- Pred pouzitim musi byt *mutex_lock* inicializovany a zamknuty volajucim vlaknom

```
int pthread_cond_signal (ptrhead_cond_t *cond)
```

- Volane z ineho vlakna
- Signalizuje prebudenie jedneho z vlakien, uspanych and podmienkovou premennou *cond*

```
int pthread_cond_broadcast (ptrhead_cond_t *cond)
```

- Signalizuje prebudenie vsetkym vlaknam cakajucim nad podm. Premennou *cond*
-

```
int pthread_cond_timedwait (ptrhead_cond_t
*cond,pthread_mutex_t *mutex_lock,const struct timespec
*abstime)
```

- Vlakno bud zobudene signalom, alebo zobudene po uplynuti castu specifikovanom v *abstime*
- Pri vzbudeni z dovodu uplynutia casu, vracia chybu *ETIMEDOUT*, a neimplikuje znovu ziskanie *mutex_lock*

Pouzitia podmienkovych premennych *

Dalsie funkcie v POSIX threads

Globalne premenne specificke pre vlakno

Problem

- Vzhladom k poziadavkom vytvarani reentrantnych a thread-safe funkcii sa programatorom zakazuje pouzivat globalne data
- Pripadne pouzitie globalnych premennych musi byt bezstavove a vykonane v kritickej sekcii
- Kladie obmedzenia na programatorov

Riesenie

- Thread specific data – **TSD**
- Globalne premenne, ktore mozu mat pre kazde vlakno inu hodnotu

Implementacia TSD

Standarne riesenie

- Pole idnexovane jednoznacnym identifikatorom vlakna
- Vlakna musia mat rozumne velke identificatory
- Snadny pristup k datam patriacim inym vlaknam – potencialne riziko nekorektneho kodu
- Vykonostny problem a sanca false sharing ak sa do pola s premennymi pristupuje prilis casto

Riesenie POSIX standard

- Identifikator(kluc) a asociovana hodnota
- Identifikator je globalny, asociovana hodnota je lokalna premenna
- Kluc – *pthread_key_t*

- Asociovana hodnota – univerzalny ukatzatel, tj void *

Posix kluce

```
int pthread_key_create (ptrhead_key_t *key,void
(*destructor)(void*))
```

- Vytvori novy kluc – jedna sa o globalnu premennu
- Hodnota asociovaneho ukazatela je nastavena na NULL pre vsetky vlakna
- Parameter *destructor* – funkcia, ktora bude nad asociovanou hodnotou vlakna volana v okamihu ukoncenia vlakna, pokial bude asociovana hodnota nenulovy ukazatel
- Parameter destructor je nepovinny, lze nahradit NULL

Pouzitie posix klucov

Znizenie kluca a asociovanych ukazatelov

- Int pthread_key_delete(pthread_key_t_ key)
- Nevola ziadne *destructor* funkcie
- Programator je zodpovedny za dealokaciu objektov pred znicenim kluca
- Funkcia destructor sa vola len ked konci vlakno a pthead_key_delete nie je null

Funkcie na ziskanie a nastavenie hodnoty asociovaneho ukazatela

- Void * pthread_getspecific (pthread_key_t key)
- ```
  int pthread_setspecific (ptrhead_key_t key,const
  void *value)
  ```

Rozne

Pthread_t pthread_self ()

- vraci unikatny systemovy identifikator vlakna – sama sebe

int pthread_equal(pthread_t thread1, pthread_t thread2)

- vraci nenula pri identite vlakien thread1 a thread2

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;int
pthread_once(pthread_once_t *once_control,void
(*init_routine)(void));
```

- prve volanie tejto funkcie z akehokolvek vlakna zposobi vykonanie kodu *init_routine*. Dalsie volania nemaju ziaden efekt
- init_routine moze byt napriklad inizialicovanie premennych nazaciatku atd

Dalsie funkcie v POSIX threads

Planovanie(scheduling) vlakien

- neni definovane, vacsinou je defaultna politika dostacujuca
```

- POSIX threads poskytuje finkcie na definiciu vlastnej politky a priorit vlakien
- Neni pozadovana implementacia tejto casti API
- Chovanie nieje garantovane

Sprava priorit mutexu

Zdielanie podmienkovych premennych medzi procesmi

Vlakna a obsluha posix signalov

Read-Write zamky

Typicke konstrukcie

Ctenari a pisari – WRRM mapa

Specifikacia problem

- Vlakna casto citaju zdielanu hodnotu, ktora je relativne menej casto modifikovana (Write-Rarely-Read-Many)
- Je chcene aby citanie hodnoty mohlo prebiehat subezne

Mozne problem

- Subezny pristup dvoch vlakien-pisatelov moze vyustit v nekonzistentne data alebo mat neziaduci vedlajsi efekt, napriklad memory leak
- Subezny pristup vlakna-pisarare v okamihu citania hodnoty inym vlaknom-ctenarom moze vyustit v citanie neplatnych, nekonzistentnych dat

Ctenari a pisari – riesenie

Riesenie s pouzitim POSIX Threads

- Citanie a modifikacia dat bude prebiehat v kritickej sekcii
- Pristup do kritickej sekcie bude riadeny pomocou funkcie pthread_*

Dalsie poziadavky

- Vlakno-ctenar moze vstupit do kritickej sekcie pokial v nej nieje alebo na nu necaka vlakno-pisar
- Vlakno-ctenar moze vstupit do kritickej sekcie pokial su v nej ine vlakna-ctenari
- Pristup vlakien-pisaru je serializovany a maju prednost pred pristupom vlakien-ctenarov

Ctenari a pisari – implementacia

Jednoduche riesenie

- Pouzit jeden pthread_mutex_t pre kriticku sekciu
- Vylucuje subezny pristup vlakien-ctenaru

Lepsie riesenei

- Implementuje novy typ zamku – rwlock_t
- Funkcia pracujuca s novym zamkom
    - Rwlock_rlock(rwlock_t *l) – vstup vlakna-ctenare
    - Rwlock_wlock(rwlock_t *l) – vstup vlakna-pisare
    - Rwlock_ulock(rwlock_t *l) – opusteni libovolnym vlaknem
- Funkcie rwlock implementovane s pouzitim podmienkovych premennych z POSIX thread API

*

Bariery

Specifikace problem

- Synchronizacni primitivum
- Vlaknu je dovolene porkacovat po bariere az ked ostatne vlakna dosiahli bariery
- Naivna implementacia cez mutexy vyzaduje aktivne cakanie – nemusi byt vzdy efektivne

Lepsie riesenie

- Implementacia bariery pouzitim podmienkovej premenej a pocitadla
- Kazde vlakno ktore dosiahne bariery zvysi pocitadlo
- Pokial nieje dosiahnuty pocet vlakien, podmienene cakanie

*

Problem

- Po dosiahnuti barery vsetkymi vlaknami, je mutex count_lock postupne zamknuty pe vsetky vlakna
- Dolny odhad na dobu behu bariery je teda O(n), kde n je pocet vlakien participujucich na bariere

Mozne riesenie

- Implementacia bariery metodou binarneho polenia
- Teoreticky dolny odhad na barieru je O(n/p + log p), kde p je pocet procesorov

Chyby, okrem nezamykaneho pristupu ku globalnej premennej

Typicke chyby – situace 1

- Vlakno V1 vytvara vlakno V2
- V2 pozaduje data od V1
- V1 plni data az po vytvoreni V2
- V2 pouzije neinicalizovane data

Typicke chyby – situace 2

- Vlakno V1 vytvara Vlakno V2
- V1 preda V2 pointer na lokalne data V1

- V2 pristupuje k datam asynchronne
- V2 pouzije data, ktora uz neexistuju(V1 skoncilo)

Typicke chyby – situace 3

- V1 ma vyssiu prioritu nez V2, citaju rovnake ata
- Neni garantovane ze V1 pristupuje k datam pred V2
- Pokial ma V2 destruktivne citanie, V1 pouzije neplatne data

Ladenie programov s POSIX vlakny

Valgrind

- Simulacia behu programu
- Analyza jedneho behu programu

Nastroje valgrindu

- Memcheck – detektce nekonzistentneho pouzita pameti
- Callgrind – jednoduchy profiler
    - Kcachegrind -visualizace
- Helgind – detekce nezamykanych pristupov ku zdielanym premennym v POSIX thread programoch

Rozsirenie POSIX threads – nepovinna dle standard -> bariery, RW zamky

Dalsie sposoby synchronizacie

Synchronizacie procesov

Problem – jak synchronizovat procesy

- Mutexy z POSIX threads dle standard sluzia iba pre synchronizaciu vramci procesu
- Pre realizaciu kritickych sekcii v roznych procesoch je treba aj inych synchronizacnych primitive
- Podpora zo strany OS

Semafory

- Citace pouzivane ku kontrole pristupov ku zdielanym zdrojom
- POSIX semafory(vramci procesu)
- System V semafory(medzi procesy)
- Daju sa vyuzit I pre synchronizaciu vlakien

Semafor

- Celociselny, nezaporny citac, ktoreho hodnota indikuje “obsadenost” zdielaneho zdroja
    - Nula – zdroj vyuzivan a neni k dispozici
    - Nenula – zdroj nevyuzivan, je k dispozici

- Sem_init() – inicializuje citac zadanou defaultnou hodnotou
- Sem_Wait() – znizi citac, pokial moze a skonci, inak blokuje
- Sem_post() – zvysi citac o 1, pripadne zobudi cakajuce vlakno

Semafory vs mutexy

- Mutex moze odomknut iba to vlakno, ktore ho zamklo
- Semafor moze byt spravovany/manipulovany roznymi vlaknami

Monitor

- Synchronizacne primitivum vyssie programovacieho jazyka
- Oznacenie kodu, ktory moze byt subezne vykovanany najviac jednym vlaknom
- JAVA – klucove slovo *synchronized*

Semafory, mtuexy a monitory

- So semaformi a mutexmi je explicitne manipulovane programatorom
- Vzajomne vyluucenie realizovane monitorom je implicitne, tj, explicitne zamykanie skrz explicitne primitive doplni prekladac

Vlakna v MS Windows

Vyssi programovaci jazyk

- C++11
- JAVA
- …

Posix Thread pre Windows

- Existuje knihovna poskytujuca POSIX thread interface

Nativne rozhranie MS Windows

- Priame systemove volania – sucast jadra OS
- Iba ramcovo podobna funkcionalita ako POSIX threads – tie iste veci riesene castokrat inak
- Na rozdiel od POSIX threads nema nepovinne casti (teda neexistuju rozne implementacie toho isteho)

Windows vs. POSIX Threads – Funkce

Windows	POSIX Threads
Pouze jeden typ `HANDLE`	Každý objekt má svůj vlastní typ (např. `pthread_t`, `pthread_mutex_t`,...)
Jedna funkce pro jednu činnost. (např. `WaitForSingleObject`)	Různé funkce pro manipulaci s různými objekty a jejich atributy.
Typově jednodušší rozhraní (bez typové kontroly), čitelnost závislá na jménech proměnných.	Typová kontrola parametrů funkcí, lepší čitelnost kódu.

Win32 vs. POSIX Threads – Synchronizace

Windows	POSIX Threads
	Mutexy Podmínkové proměnné Semafory
Signalizace pomocí událostí.	Signalizace v rámci podmínkových proměnných.
Jakmile je událost signalizována, zůstává v tomto stavu tak dlouho, dokud ji někdo voláním odpovídající funkce nepřepne do nesignalizovaného stavu.	Signál zachycen čekajícím vláknem, pokud takové není, je signál zahozen.

Win32 vs. POSIX Threads – Základní mapování

Windows	POSIX Threads
`CreateThread`	`pthread_create` `pthread_attr_*`
`ThreadExit`	`pthread_exit`

`WaitForSingleObject`	`pthread_join`
	`pthread_attr_setdetachstate`
	`pthread_detach`

`SetPriorityClass`
`SetThreadClass`
	Pouze nepřímo mapovatelné.
	`setpriority`
	`sched_setscheduler`
	`sched_setparam`
	`pthread_setschedparam`
	...

Win32 vs. Linux/UNIX – Mapování synchronizace

Windows	Linux threads	Linux processes
Mutex	PThread Mutex	System V semafor
Kritická sekce	PThread Mutex	—
Semafor	PThread podm. proměnné POSIX semafor	System V semafor
Událost	PThread podm. proměnné POSIX semafor	System V semafor

Win32 vs UNIX – pozorovanie

Pozice vlakien v MS Windows

- Silnejsie postavenie nez vlakna v Linuxe
- Synchronzacne prostriedky funguju I medzi procesmi
- Vlakna vo vlakne(Processes-Threads-Fibers)
    - Scheduling Fibers neriesi OS, ale kooperativne – vlakna sa sami rozhoduju ci su na rade a podobne, chyba externy zasach OS

- User-mode-scheduling(UMS) – kooperativne planovanie

Vyhodny len jedneho typu zastupujuceho objekty

- Jedinou funkciou sa da cakat na nekonkretne vlakno
- Jednou funkciou sa da cakat na vlakno a na mutex

05 – implementacia lock-free datovych struktur

Snazime sa vyhnut pouzitiu zamykacich primitive pre zvysenie vykonu

Klasicka skola viacvlaknoveho programovania

- Pristup k zdielanym datam musi byt chraneny
- Pristupy k datam sa musia serializovat s vyuzitim roznch synchronizacnych primitive(mutexy,semafory,monitory_
- Vlakna operuju s datami tak, aby sa tieto operacie javili ostatnym vlaknam ako atomicke operacie

Problemy

- Omeskanie pri pristupu k zdielanym datam
- Uviaznutie, zivost, ferovost
- Korektnost implementacie
    - Tazko dokazat a tazko nevieme povedat na vyssej urovni ci sa jedna o atomicku operaciu
- Atomicita operacii

Lock-free programovanie

- Programovanie paralelnych aplikacii bez pouzitia zamykania alebo inych makro-synchronizacnych mechanizmov

Vlastnosti lock-free programovania

- **Vyzaduje HW podporu**
- Pouziva sa typicky jedna jedina atomicka konstrukcia/instrukcia – napr compare, swap
- Minimalne prodlevy suvisiace s pristupom k datam
- Neexistuje uviaznutie, je garantovana zivost(program stale bezi, nenastal deadlock, ale to neznamena ze nehladovie/nestarne)
- Algoritmicky obtiaznejsie uvazovanie
- Korektnost algoritmu nachlna na optimalizace prekladace

Wait-free procedura

- Procedura, ktora bez ohladu na subeh dvoch ci viac vlakien, dokonci svoju cinnost v konecnom casu, tj neexistuje subeh ktory by until procedure, nekonecne dlho cakat, ci vykonavat nekonecne vela operacii -> nehladovie, nestarne

Lock-free procedura

- Procedra ktora garantuje ze pri lubovolnom subehu vela superiacich vlakien, vzdy aleaspon jedno vlakno uspesne dokonci svoju cinnost. Niektore superiace vlakna mozu byt lubovolne dlho nutene odkladat dokoncenie svojej cinnosti

Priklad instrukcii nevhodnych k budovaniu lock-free dat. Struktur pre viacvlakn. aplikacie

- Test-and-set
- Swap
- fetch-and-add
- Fronty s atomickymi operaciami vlozenia a vyberu

Su vhodne napriklad pre konstrukciu CAS

*

CAS – compare and swap

- Porovnava obsah specifikovanej pametovej adresy *addr* s ocakavanou hodnotou *exp* a vpripade rovnosti nahradi obsah pametovej adresy novou hodnotou *val*
- O uspechu ci neuspechu informuje uzivatela navratovou hodnotou
- Cela procedura prebehne **atomicky a viditelne pre ostatne vlakna**

Princip pouzitia instrukcie CAS

Postup pri pristupe k zdielanym datam

- Precitaj stavajucu hodnotu zdielaneho objektu
- Pripravim novu hodnotu zdielaneho objektu
- Aplikuj instrukciu CAS

Navratova hodnota

- **True** -> object nebol v medzicase modifikovany, novo vypocitana hodnota je platna a ulozena v zdielanom objekte
- **False** -> object bol v medzicase modifikvoany (z ineho vlakna), instrukcia CAS nemala ziaden efekt a je nutne cely postup opakovat

Nebezpecie pouzita CAS – ABA problem

Klucova vlastnost

- Modifikacia objektu ktora pebehla medzi nacitanim hodnoty objektu a aplikacie instrukcie CAS nesmie vyprodukovat tutiez hodnotu zdielaneho objektu

Mozny chybovy scenar

- Hodnota objektu, nacitana vlaknom A za ucellom pouzitia v naslednej nstrukcii CAS, je x
- Pred pouzitim instrukcie CAS vlaknom A, je object modifikovana inymi vlaknami, tj nabyva hodnoty rozne od x
- V okamziku aplikacie instrukcie CAS vlaknom A, ma object opet hodnotu x
- Vlakno A nespozna ze sa hodnota objektu menila
- Nasledna aplikacia instrukcie CAS uspeje

CAS – vykonavanie instrukcii mimo poradie

- Pokial pouzivame CAS na zpristupnenie nejakych dat, je potreba zaistit aby predchadzajuce inicializacie premennych boli uz v okamziku vykonania instrukcie CAS vykonane
- Vyzaduje pouzitie pametovej bariery
- Dotknute premenne musia byt oznacene ako nestale

CAS – cena

- Pouzitie cas odstranilo reziu suvisiacu so zamykanim
- Zustava vsak rezia suvisiaca s coherenciou cache pamete

Programovanie s CAS

Win32

- InterlockedCompareExchange()

Assembler i386(pre x86_64, nutne premenovat edx na rdx)

```
inline int32_t compareAndSwap(volatile int32_t & v, int32_t exValue, int32_t cmpValue){ asm volatile ("lock; cmpxchg :%%ecx,(%%edx)" : "=a" (cmpValue) :"d" (&v), "a" (cmpValue), "c" (exValue));return cmpValue;}
```

GCC – zaudovane funkcie

- `bool __sync_bool_compare_and_swap (T *ptr, T old, T new,...)`
- `T __sync_val_compare_and_swap (T *ptr, T old, T new, ...)`

WRRM mapa = priklad

Write Rarely Read Many Mapa

- Zprostredkovava preklad jednej entity na inu(Kluc->hodnota)
- Priklad – kurz koruny k inym menam
  - Meni sa raz denne
  - Pouziva sa pri kazdej transakcii

Mozne implementacie s vyuzitim STL

- Map, hash_map

- assoc_vector(usporiadane dvojice)

Pouzitie

- Map<Key,Value> mojeMapa;

Mutex implementacia *

Implementacia s pouzitim instr. CAS

Operacia citania

- Prebieha zcela bez zamykania

Operacia zapisu

- Vytvoenie kopie stavajucej mapy
- Modifikacia/pridanie dvojice do vytvorenej kopie
- Atomicka zamena nvoej verzie mapy za predchadzajucu

Realne omedzenia CAS

- Obecne pouzitie schemata CAS na WRRMMap by vyzadovalo atomicku zmenu relativne rozsiahlej olasti pameti
- HW podpora pre CAS je obmedzena na niekollko bytov(typicky jedno, az dva slova procesoru)
- Atomicku zamenu vykoname cez ukazatel

*

WRRM map – vlastnosti a problem dealokace

Preco je nutna instrukcia CAS a nestaci len p0ld = pNew

- Interferuju vlakna pisary
- Vlakno A urobi kopiu mapy
- vlakno b udela kopiu mapy, vlozi novy kluc a dokonci operaciu
- vlakno A vlozi novy kluc
- vlakno A nahradi ukazatel, vsetko co vlozilo B je ztratene

Update

- je lock-free, ale neni wait-free

Sprava pameti

- ***Update nemoze uvolnit staru kopiu datovej struktury, ine vlakno moze nad datovou strukturou vykonavat operaciu citania***
- Mozne riesenie: Garbage collector(JAVA)

Mozne riesenie v jazykoch bez garbage collector – odlozena dealokace pameti

- Miesto delete sa spusti(asynchronne) nove vlakno
- Nove vlakno pocka 200ms a vykona dealokaciu

Myslienka

- Nove operacie prebiehaju nad novou kopiou, za 200ms sa vsetky zapocate operacie nad starou kopiou dokonci a bude bezpecne strukturu dealokovat

Problemy

- Kratkodobe intenzivni prepisovani hodnot alebo vkladanie novych hodnot moze sposobit netrivialne pametove naroky
- Neni garantovane ze sa vsetky operacia citania z inych vlakien za dany casovy limit dokoncia

Napad

- Napodobime metodu pouzivanu pri automatickom uvolnovani pameti k tomu, aby sme mohli explicitne dealokova strukturu
- Pocitanie odkazov – s kazdym ukazatelom je zviazane cislo, ktore udava pocet vlakien, ktory tento odkaz este pouzivaju

Modifikacia WRRM mapy

- Procedura *Update* vykona podmienenu dealokaciu, tj, dealokuje object odkazovany ukazatelom, iba pokial ziadne ine vlakno ukazatel nepouziva
- Procedura *Lookup* postupuje tak, ze zvysi citac spojeny s ukazatelom pristupi k structure skrz tento ukazatel, znizi citac po ukonceni prace so strukturou a podmienene dealokuje strukturu

Citac asociovany s ukazatelom MAP<K,V>*

*****

- CAS instrukcia nad ukazatelom pData_

- podmienena dealokacia:

    If (pData_->second==0) delete (pData_->first);

Problem v procedure Lookup

- Vlakno A *nacita strukturu* Data(cez *pDate_) a je prerusene
- Vlakno B vlozi kluc, znizi citac a dealokuje *pOld->first
- Vlakno A *zvysi citac*, ale pristupi k neplatnemu ukazatelu

WRRM Map a pocitanie odkazov – CAS2

Problem predchodzej verzie

- Akcia uchopenia ukazatela a zvysenia odpovedajuceho citaca neboli atomicke

Riesenie

- Pomocou jednej instrukcie CAS je treba prepnut ukazatel a koektne manipulovat s citacom
- Teoreticky je mozne implementovat CAS pracujuci s viacero strukturami zaroven, samozrejme straca sa efektivita pokial neexistuje HW podpora
- Modern procesory maju podporu pre instrukciu CAS pracujucu s dvoma po sebe ulozenymi slovami procesoru – **CAS2**

*

WRRM Map s vyuzitim CAS2

- Struktura *Data* je tvorena dvoma slovmi *ukazatel* a *citac*
- Ukazatel a citac su lozene v pameti vedla seba
- Strukturu je mozne modifikovat pomocou instrukcie CAS2

*

WRRM Map s vyuzitim CAS2 – stale nefunkcne

Otazka

- Dokazeme atomicky realizovat pocitanie odkazov, je teda navrhovane riesenie korektne?

Problem

- Zvysenie a znizenie citaca procedurou lookup je v zcela nezavislych blokoch. Pokial sa medzi vykonanim tychto blokov realizuje nejaka procedura Update, tak pripocitanie a odcitanie jednotky k citacu prebehne nad inymi ukazatelmi
- Riziko predcasnej dealokacie
- *Ztrata ukazatelov na stare kopie – memory leak*

Riesenie

- Citac spojeny s ukazatelom pouzijeme ako straz
- Procedura update bude vykonavat zmeny struktury len ak ziadne ine vlakno k structure nepristupuje

WRRM Map s vyuzitim CAS2 – realizacia Update

Odkladanie vykonania modifikacie v procedura *Update*

- Atomicke nahradenie ukazatela sa deje v okamihu, kedy su vsetky ostatne vlakna mimo procedure Lookup
- Casove intervaly, po ktorych sa jednotlive vlakna nachadzaju v procedure Lookup ctenarum sa vsak mozu prekryvat
- Citac po celu dobu existencie ineho vlakna v procedure *Lookup* neklesa na minimalnu hodnotu a procedra *Update* tzv hladovie(Starve)

Optimalizacia procedury *Update*

- pri opakovanych nesuspechoch instrukcie CAS dochadza k opakovanemu kopirovaniu povodnej struktury a nasledneho mazania vytvorenej kopie
- neefektivne opakovane kopirovanie lze odstranit pomocou pomocneho ukazatela *last*

*

WRRM Map – pozorovanie ohladom realizacie s CAS2

Lookup

- neni wait-free, inkrementacia a dekrementacia citaca interferuje s procedurou update
- volanie procedur update je malo – nevadi

Update

- neni wait-free, interferuje s procedurou Lookup
- volanie procedur lookup je mnoho – problem

Coho sme dosiahli

- WRRM BNTM Mapa
- **Write Rarely Read Many, But Not Too Many**

Dalsie programatorske rozhrania

MCAS

- Rozsirenie standardnej instrukcie CAS pre pouzitie s lubovolne velkou datovou strukturu

Transakcna pamet

- Pamet je modifikovana v jednotlivych transakciach
- Transakcia seskupuje vela cteni a zapisu do pameti – je schopna obsiahnut komplexnu modifikaciu datovych struktur
- Zakladnym manipulovatlnym objektom je slovo procesoru, tj obsah jednej pametovej bunky
- Priklad: presun prvku v dynamicky zretazenom zozname

Load-Link(LL)/Store-Conditional(SC)

- Dvojca instrukcia ktora dohromady realizuje CAS
- LL nacita hodnotu a SC ju prepise, ale len pokial nebola modifikovana, pricom za modifikaciu sa povazuje aj prepsanie na tutez hodnotu
- LL/SC stejna sila ako CAS, avsak nema ABA problem
- HW podpra: Alpha,PowerPC,MIPS,ARM

Problemy

- Zmena kontexu sa vpraxi poazuje za modifikaciu miesta

- Teoeticky neni mozne realizovat wait-free procedure
- Narocne ladenie

Navrh Lock-free datovych struktur

- Je mozne navrhnut lockfree datove struktury
- Zaujimava algoritmika
- Obtiazne pokial chceme deterministicke uvolnovanie pameti
- Vhodne pre prostedie s garbage collectorom

06 – pokrocile rozhrania pre impl. Paral. Aplikacii

Iny sposob programovania v prostredi so zdielanou pametou

Nevyhody POSIX threads a lock-free pristupu

- Na prilis nizkej urovni
- Vhodne pre systemovych programatorov
- Prilis zlozity pristup na riesenie jednoduchych veci

Co by sme chceli

- Paralelna konstrukcia na urovni programovacieho jazyka
- Prostriedok vhodny pre aplikacnych programatorov
- Lahke vyjadrenie bezne pouzivanych paralelnych konstrukcii

OpenMP

Myslienka

- Sejmout z programatora bremeno nizkourovnove implementace
- Programator sa sustredi na to, co chce v programe urobit, nema sa zabyvat tym ako sa to stane

Realizace

- Programator informuje prekladac o zamyslanej paralellizacie uvedenim *znaciek v zdrojovom kode* a oznacenim blokov
- Pri prekladu prekladac sam doplni nizkourovnovu realizaciu paralelizacie

OpenMP ponuka

- Pragma direktivy prekladaca
    - #pragma omp *direktiva [seznam klauzuli]*
- Knihovnove funkcie
- Premenne prostredia

Prekladac kodu

- Prekladac podporujuci standard OpenMP
    - Pri preklade pomocou GCC je nutna volba -fopenmp

    - G++ -fopenmp myapp.c
- Podporovane najpouzivanejsimi prekladacmi(aj visual c++)
- Mozno prelozit do sekvencneho kodu –> odignoruje znacky pre paralelny kod ak sa nepouzije prepinac

*

Direktiva *parallel*

Pouzitie

- Strukturovany blok, tj {..}, nasledujuci sa touto direktivou sa prevedie paralelne
- Mim paralelne bloky sa kod vykonava sekvencne
- Vlakno ktore narazi na tuto direktivu sa stave hlavnym vlaknom(master) a ma identifikaciu vlakna rovnu 0

Podmienene spustenie

- Klauzula if(vyraz typu bool)
- Ak sa vyhodnoti vyraz na false direktiva *parallel* sa ignoruje a nasledujuci blok je vykonany iba v jednej kopii

Stupen paralelizmu

- Pocet vlakien
- Prednastaveny pocet specifikovany premennou prostredia
- Klauzula *num_thread*(vyraz typu int)

Direktiva *parallel* – datova lokalita

Klauzula *private (seznam premennych)*

- *Vymenovane premenne sa zduplikuju a stanou sa lokalnymi promennymi v kazdom vlakne*

Klauzule firstprivate (seznam premennych)

- Viz private, s tym, ze vsetky kopie premennych su inicializovane na hodnotu originalnej kopie

Klauzule *shared (zoznam premennych)*

- Vymenovane premenne budu explicitne existovat v iba jednej kopii
- Pristup k zdielanym premennym nutne serializovat

Klauzule *default ([shared|none])*

- *Shared*: vsetky premenne su zdielane, pokial neni uvedene inak
- *None*: vynucuje explicitne uvedenie kazdej premennej v klauzuli private alebo v klauzuli shared

Direktiva *parallel* – redukce

Klauzule *reduction (operator: seznam premennych)*

- Pri ukonceni paralelneho bloku su vymenovane private premenne zkombinovane pomocou uvedeneho operatoru
- Kopie uvedenych premennych, ktore su platne po ukonceni paralelneho bloku, su naplnene vyslednou hodnotou
- Premenne musia byt skalarneho typu(nesmie byt pole, struktury, etc.)
- Pouzitelne operatory: +,*,&,|,^,&& a ||

Direktiva *for*

Pouzitie

- Iteracia nasledujuceho for cyklu budu prevedene paralelne
- Musi byt pouzite vramci bloku za direktivou *parallel*(inak prebehne sekvencne)
- Mozny zkrateny zapis #pragma omp parallel for

Klauzule *private, firstprivate,reduction*

- Stejne ako pre direktivu parallel

Klauzule *lastprivate*

- Hodnota privatnej premennej vo vlakne zpracovavajucu poslednu iteraciu for cyklu je ulozena do kopie premennej platnej po skonceni cyklu

Klauzule *ordered*

- Bloky oznacene direktivou ordered v tele paralelne provadeneho cyklu su prevedene v tom poradi, v akom by boli prevedene sekvencnym programom
- Klauzule ordered je povinna, pokial telo cyklu obsahuje ordered bloky

Klauzule *nowait*

- Jednotlive vlakna sa nesnchronizuju po vykonani cyklu

Klauzule *schedule (typ planovani [,velkost])*

- Urcuje ako budu iteracie rozdelene/mapovane medzi vlakna
- Implicitne planovanie je zavisle na implementacii

Direktiva *for* – planovanie iteracii

Static

- Iteracie cyklu rozdelene do blokov o specifickej velkosti
- Bloky staticky namapovane na vlakna(round robin)

    - Round robin – priradi procesu/vlaknu kvantum casu, po ktore moze byt process/vlakno na procesore. Po ubehnuti casu je spustene ine vlakno/proces
- Pokial neni uvedena velkost, iteracie rozdelene medzi vlakna rovnomerne(ak je to mozne)

Dynamic

- Bloky iteracii cyklov v pocte specifikovanym parametrom *velkost* pridelovane vlaknam na ziadost, tj v okamihu kedy vlakno dokoncilo svoju predchadzajucu pracu
- Defaultna velkost bloku je 1

Guided

- Bloky iteracii maju velkost proporcionalnu k poctu nezpracovanych iteracii vydelenym poctom vlakien
- Specifikovana velkost ***k*** udava minimalnu velkost bloku(defaultne 1)
- *Priklad:*
    - K=7, 200 volnych iteracii, 8 vlakien
    - Velkost bloku: 200/8=25,175/8=21, ⋯, 63/8=7, ⋯

Runtime

- Typ planovania urceny az za behu premennou *OMP_SCHEDULE*

Direktiva *sections*

Pouzitie:

- Strukturovane bloky, kazdy oznaceny direktivou *section*, mozu byt vramci bloku oznacenym direktivou *sections* vykonavane paralelne
- Mozny zkrateny zapis #pragma omp parallel sections
- Umoznuje definovat rozny kod pre rozna vlakna

Klauzule private, firstprivate, reduction, nowait

- Rovnake ako v predoslych pripadoch

Klauzule *lastprivate*

- Hodnoty privatnych premennych v poslednej sekcii(podla zapisu kodu) budu platne po skonceni bloku *sections*

*

Vnorovanie direktiv *parallel*

Nevnoreny paralelizmus

- Direktiva parallel urcuje vznik oblasti paralelneho vykonavania

- Direktivy for a sections urcuje ako bude praca mapovana na vlakna podla rodicovskej direktivy parallel

Vnoreny paralelizmus

- Pri nutnosti paralelizmu vramci paralelneho bloku, je treba znovu uviest diektivu parallel
- Vnorovanie je podmienene nastavenim premennej prostredia OMP_NESTED(hodnoty TRUE, FALSE)
- Typicke pouzite vnorene for-cykly
- Obecne vnorovanie direktiv v OpenMP pomerne komplikovane
- Moze vzniknut viac vlakien nez CPU jadier – system priliz zatazeny tym ze strieda vlakna na jadrach a nie je teda vhodne vnoreny paralelizmus vzdy pouzivat

Direktiva *barrier*

Bariera

- Miesto, ktore je dovolene prekrocit az k nemu dorazia vsetky ostatne vlakna
- Direktiva bez klauzuli, tj #pragma omp barrier
- Vztahuje sa k strukturalne **najblizse**j directive *parallel*
- Musi byt volane **vsetkymi** vlaknami v odpovidajucom bloku direktivy *parallel*
    - Nefunguje teda zavolat na napriklad 2 zo 4
    - Da sa obist tym ze sa v parallel bloku vytvori blok na 2 vlakna, kde uz to bude fungovat

Poznamka ku kodovaniu

- Direktivy prekladace niesu sucastou jazyka
- Je mozne ze v ramci prekladu bude vyhodnoteny blok, v ktorom je umiestnena direktivy barrier, ako nevykonatelny blok a odpovedajuci kod nebude vo vyslednom spustitelnom suboru vobec priomny
- Direktivu *barrier* je nutne umiestnit v bloku, ktory sa bezpodmienecne vykona(zodpovednost programatora)

Direktiva *single* a *master*

Direktiva *single*

- V kontexte paralelne vykonavaneho bloku je nasledujuci strukturalny blok vykonany iba jednym vlaknom, pricom neni urcene ktorym

Klauzule *private, firstprivate*

Klazule *nowait*

- Pokial neni uvedena, tak na konci strukturalneho bloku oznaceneho direktivou *single* je vykonana bariera

Direktiva *master*

- Specialny pripad direktivy single
- Tym vlaknom, ktore vykona strazeny blok, bude hlavne(master) vlakno

Direktiva *critical a atomic*

Direktiva *critical*

- Nasledujuci strukturovany blok je chapany ako kriticka sekcia a moze byt vykonany maximalne jednym vlaknom v danom case
- Kriticka sekcia moze byt pomenovana, subezne je mozne vykonavat kod v kritickyh sekciach s inym nazvom
- Pokial neni uvedene inak, pouzije sa implicitne meno
- #pragma omp critical [(name)]

Direktiva *atomic*

- Nahradzuje kriticku sekciu nad jednoduchymi modifikaciami(updaty) premennych v pameti
- Atomicita sa aplikuje na jeden nasledujuci vyraz
- Obecne vyraz musi byt jednoduchy(len *load a store*)
- Neatomizovatelny vyraz: x =y =0 -> dve ulozenia do pameti

Direktiva *flush*

Problem(nestale premenne)

- Modifikacie zdielanych premennych v jednom vlakne moze ostat skryta ostatnym vlaknam

Riesenie

- Explicitna direktiva pre kopirovanie hodnoty premennej z registru do pameti a zpet
- #pragma omp flush [(seznam)]

Pouzitie

- Po zapise do zdielanej premennej
- Pred citanim obsahu zdielanej premennej
- Implicitne v miestach bariery a konca blokov(pokial niesu bloky v rezime nowait)

Direktiva *threadprivate* a *copyin*

Problem(thread-private data)

- Pri statickom mapovani na vlakna je drahe pri opakovanom vznku a zaniku vlakien vytvarat kopie privatnych premennych
- Obcas chceme private globalne premenne

Riesenie

- Perzistentne private premenne – preziju zanik vlakna
- Pri znovuvytvoreni vlakna sa premenne znovupouziju
- #pragma omp threadprivate (seznam)

Obmedzenia

- Nesmie sa pouzit dynamicke planovanie vlakien
- Pocet vlakien v paralelnych blokoch musi byt zhodny

Direktiva *copyin*

- Jako threadprivate, ale s inicializaciou
- Viz private versus firstprivate

OpenMP knihovnove fukcie – pocet vlakien

```
void omp_set_num_threads (int num_threads)
```

- Specifikuje kolko vlakien sa vytvori pri dalsej directive parallel
- Musi byt pouzite pred samotnou konstrukciou parallel
- Je prebito klauzulou num_threads, pokial je pritomna
- Musi byt povolene dynamicke modifikovanie procesov(OMP_DYNAMIC, omp_set_dynamic())

```
int omp_get_num_threads ()
```

- Vracia pocet vlakien v teame strukturalne najblizsej direktivy parallel, pokial neexistuje, vracia 1

OpenMP knihovnove fukcie – pocet vlakien a procesorov

```
int omp_get_max_threads ()
```

- Vracia maximální počet vláken v týmu.

```
int omp_get_thread_num ()
```

- Vracia unikátní identifikátor vlákna v rámci týmu.

```
int omp_get_num_procs ()
```

- Vracia počet dostupných procesorů, které mohou v daném okamžiku participovat na vykonávání paralelního kódu.

```
int omp_in_parallel ()
```

- Vracia nenula pokud je voláno v rozsahu paralelního bloku.

OpenMP knihovnove fukcie – kontrola vytvarania vlakien

```
Void omp_set_dynamic (int dynamic_threads)
```

```
int omp_get_dynamic()
```

- Nastavuje a vracia, ci je programatorovi umoznene dynamicky menit pocet vlakien vytvorenych pri dosiahnuti direktivy parallel
- Nenulova hodnota dynamic_threads znaci **povoleno**

```
void omp_set_nested (int nested)
```

```
int omp_get_nested()
```

- Nastavuje a vracia ci je povoleny vnoreny paralelizmus
- Pokial neni povolene, vnorene paralelne bloky su serializovane

OpenMP knihovnove fukcie – mutexy

```
void omp_init_lock (omp_lock_t *lock)
```

```
void omp_destroy_lock (omp_lock_t *lock)
```

```
void omp_set_lock (omp_lock_t *lock)
```

```
void omp_unset_lock (omp_lock_t *lock)
```

```
int omp_test_lock (omp_lock_t *lock)
```

```
void omp_init_nest_lock (omp_nest_lock_t *lock)
```

```
void omp_destroy_nest_lock (omp_nest_lock_t *lock)
```

```
void omp_set_nest_lock (omp_nest_lock_t *lock)
```

```
void omp_unset_nest_lock (omp_nest_lock_t *lock)
```

```
int omp_test_nest_lock (omp_nest_lock_t *lock)
```

- Inicializuje, ničí, blokujuco čeká, odemyká a testuje
- – normálny a rekurzivny mutex.

Premenne prostredia

OPM_NUM_THREADS

- Specifikuje defaultny pocet vlakien, ktore sa sa vytvoria pri pouziti direktivy parallel

OPM_DYNAMIC

- Hodnota TRUE, umoznuje za behu dynamicky menit pocet vlakien

OMP_NESTED

- Povoluje hodnotou TRUE vnoreny paralelizmus
- Hodnotou FALSE specifikuje ze vnorene paralelne konstrukcie budu serializovane

OMP_SCHEDULE

- Udava defaultne nastavenie mapovania iteracii cyklu na vlakna
- Priklady hodnot: “static, 4”, dynamic, guided

Intel’s Thread Building Blocks (TBB)

Co je Intel TBB

- TBB je c++ knihovna pre vytvaranie viacvlaknovych aplikacii
- Zalozena na principu zvanom *generic programming*
- Vyvynute synergickym spojenim Pragma direktiv(OpenMP), standarnej knihovne sablon(STL, STAPL) a programovacich jazyku podporujucich pracu s vlaknami(Threaded-C, Cilk)

Generic Programming

- Vytvaranie aplikacii specializaciou existujucich predpripravenych obecnych konstrukcii, objektov a algoritmov
- Da sa najst v objektovo orientovanych jazykoch – C++, JAVA
- V c++ su obecnou konstrukciou sablony(Templates)
    - Queue<int>
    - Queue<Queue<Char>>

Pouzitie TBB

Vlastnosti Intel TBB

- Knihovna, implementovana s vyuzitim standardneho c++
- Nepozaduje podporu specialneho jazyka ci prekladaca
- Podporuje vnoreny paralelizmus, potazmo je mozne stavat zlozitejsie paralelne systemy z mensich paralelnych component
- Cielom pouzitia je nechat programatora specifikovat ulohy k paralelnemu vykonaniu, nie nutit ho popisovat co a ako robia jednotlive vlakna

Moznosti TBB

TBB poskytuje sablony pre

- Paralelizaciou iteracii jednotlivych cyklov – datovy paralelizmus
- Definiciu vlastnych paralelne pristupovanych datovych struktur
- Vyuzitie nizkourovnovych HW primitiv
- Zamykanie pristupov do kritickej sekcie v roznych podobach

- Lahku definiciu paralelnych subeznych uloh
- Skalovatelnu alokaciu pameti

Princip datovej paralelizacie v TBB

Paralelny for-cyklus

- Je dana mnozina nezavislych indexov, tzv rozsah(range)
- Pre kazdy index z mnoziny je vykonane telo cyklu

Paralelny for-cyklus v TBB

- Sablona, ktora ma dva parametry – rozsah a telo cyklu
- Sablona zaisti vykonanie tela cyklu pre vsetky indexy v specifikovanom rozsahu
- Rozsah je deleny na pod-rozsahy. Paralelizmu dosiahnute vykonavanim tela cyklu nad jednotlivymi pod-rozsahmi

*

Koncept delenia

- Instancie niektorych tried je nutne za behu (rekurzivne) delit
- Zavadza sa novy typ konstruktoru, deliaci konstruktor:
  - X::X(X& x, split)
- Deliaci konstruktor rozdeli instanciu triedy X na dve casti, ktore dohromady davaju povodny object. Jedna cast je priradena do x, druha cast je priradena do novo vzniknutej instancie
- Schopnost delit sa musia mat najma rozsahy, ale taktiez triedy ktorich instancie bezia paralelne a pritom nejakym sposobom interaguju, napr triedy realizujuce paralelnu redukciu

Split

- Specialna trieda definovana za ucelom odlisenia deliaceho konstruktoru od kopirovacieho konstruktoru

Koncept rozsahu

Poziadavky na triedu realizujucu rozsah

- Kopirovaci konstruktor
  - R::R (const R&)
- Deliaci konstruktor
  - R::R (constR&, split)
- Destruktor
  - R::~R ()
- Test na prazdnost rozsahu
  - Bool R::empty() const
- Test na schopnost dalsieho rozdelenia

- Bool R::is_divisible() const

Preddefinovane ssablony rozsahov

- Jednodimenzionalne: blocked_range
- Dvojdimonzionalne: blocked_range2d

TBB: blocked range

Blocked_range

- Template<typename Value> class blocked_range;
- Reprezentuje nadalej delitelny otvoreny interval [I,j)

Poziadavky na triedu *Value* specializujucu *blocked_range*

- kopírovací konstruktor
  - Value::Value (const Value&)
- Destruktor
  - Value:: ~Value ()
- Operátor porovnání
  - bool Value::operator<(const Value& i, const Value& j)
- Počet objektů v daném rozsahu (operátor–)
  - size_t Value::operator-(const Value& i, const Value& j)
- k-tý následný objekt po i(operátor+)
  - Value Value::operator+(const Value& i)

Pouztie blocked_Range<Value>

- Najdolezitejsou metodou je konstruktor
- Konstruktor specifikuje interval rozsahu a velkost najvetsieho dalej nedelitelneho sub intervale
- Blocked_range(Value begin, Value end, [size_t gransize])

Typicka specializacia

- Blocked_range <int>
- Priklad blocked_range<int>(5,17,2)
- Priklad blocked_range<int>(0,11)

TBB: parallel_for

Parallel_for<Range, Body>

- `template<typename Range, typename Body>`
- `void parallel_for( const Range& range, const Body& body);`

Poziadavky na triedu realizujuce telo cyklu

- konstruktor
- destructor

- aplikator tela cyklu na dany rozsah – operator()

TBB: parallel_reduce

Parallel_Reduce<range,body>

Poziadavky na triedu realizujuce telo redukcie

- deliaci konstruktor
- destructor
- funkcia realizujuca redukciu nad danym rozsahom – operator()
- funkcia realizujuca redukciu hodnot z roznyh rozsahov

Moznosi delenia

Trieda *Partitioner*

- paralelne konstrukcie maju treti volitelny parameter, ktory specifikuje strategiu delenia rozsahu
- parallel_for<range,body,partitioner>

Preddefinovane strategie

- simple_partitioner
  - rekurzivne deli rozsah az na dalej nedlitelne intevaly
  - pri pouziti blocked_range je volba grainsize klucova pre vyvazeni potencialu a rezie paralelizacie
- auto_partitioner
  - automaticke delenie, ktore zohladnuje zatazenie vlakien
  - pri pouziti blocked_ane voli rozsahy vacsie nez je grainsize a tieto deli iba do tej doby, nez je dosiahnute rozumne vyvazenie zataze. Volba minimalnej velkosti grainsize nesposobi nadbytecnu reziu spojenu s paralelizaciou

Paralelne pristupovane kontajnery – *vector* a *queue*

Concurrent_queue

- template<typename T> concurrent_queue
- Fronta, ku ktorej moze subezne pristupovat viacero vlakien
- Velkost fronty je dana poctom operacii vlozenia bez poctu operacie vyberu, zaporna hodnota znaci cakajuce operacie vyberu
- Definuje sekvencne iteratory, neodporuca sa ich pouzivat

Concurrent_vector

- Template<typename T>concurrent_vector

- Zvacsovatelne pole prvkov, ku ktorym je mozne subezne pristupvoat z viacero vlakien a vykonavat subezne zvacsovania pola a pristup k uz ulozenym prvkom
- Nad vektorom sa da definovat rozsah a vykonavat skrz nej paralelne operacie s prvkami ulozenymi v poli

Paralelne pristupovane kontajnery – hash_map

Concurrent_hash_map

- Mapa, v korej je mozne paralelne hladat, mazat a vkladat

Poziadavky na triedu HashCompare

- Kopirovaci konstruktor
- Destructor
- Test na ekvivalenciu objektov
- Vypocet hodnoty hashovacej funkcie

Objekty pre pristup k datam v *concurrent_hash_map*

- Pristup k parom kluc-hodnota je skrze pristupovacie triedy
- *Accessor* – pre pristup v rezime read/write
- *Const_accessor* – pre pristup iba v rezime read
- Pouzitie pristupovacich objektov umoznuje korektny paralelny pristup k zdielanym datam

*

Metody pre pracu s *concurrent_Hash_map*

- Find – dva typy -> pre konstantny accessor a normalny accessor(da sa modifikovat)
- Insert
- erase

Dalsie sposoby pouzitia

- Iteratory pre prechadzanie mapy
- Daju sa definovat rozsahy a s nimi pracovat paralelne

C++11

C++11 a vlaknovanie

Pozorovanie

- C++11 ma definovane prikazy pre podporu vlakien
- Neni treba pouzit externe knihovne ako je POSIX Thread

Ako je to mozne

- C++11 definuje virtualne viacjadrovy vypocetny stroj a teda rata sa s tym ze jazyk bude bezat na viacjadrovom vypocetnom stroji
- Veskera semantika prikazov sa odkazuje na tento virtualny vypocetny stroj
- Prikazy s podporou vlakien mozu byt sucastou jazyka
- Prenos semantiky z virtualneho vypocetneho stroja na realny HW je na zodpovednosti prekladaca

Vlakny a mutexy v c++11 *

Zamykanie v c++11

Potencionalne riziko uviaznutia

- Jazyk s plnou podporu mechanizmu vynimok
- Vyvolani vynimky v okamihu ked je vlakno v kritickej sekcii(uvnit mutexu) pravdepodobne sposobi, ze nebude vlaknem volana metoda odomykajuca zamek spojeny s kritickou sekciou

Reseni

- Vyuzitie principu RAII(**Resource acquisition is initialization**) a OOP
- Zamcenie mutexu realizovane vytvorenim lokalnej instancie hodnej preddefinovanej zamykacej triedy
- Odomknutie umiestnene do destruktoru tejto triedy
- Destructor je vykonany v okamihu opustenia rozsahu platnosti daneho objektu

RAII Zamykanie v c++11

Trieda *lock_guard*

- Obalenie standardneho zamku v RAII style
- Mutex na pozadi nejde predat inemu vlaknu, nevhodne pre podmienkove premenne
- Priklad pouzitia *

Trieda unique_lock

- Obecnejsi predatelne RAII obalenie mutexu
- Doporucene pre pouzitie s podmienkovymi premennymi

Podporovane aspekty

Podpora vlaknovania v c++11

- Vlakna
- Mutexy a RAII zamky
- Podmienene premenne
- Zdielane futures(miesta ke sa ulozia dosial nespocitane hodnoty)

Atomicita zapisu

Neatomicky

Int x,y;

Thread 1

X=17

Y=37

Thread 2

Cout << y << " ";

Cout << x << endl;

Nema definovane chovanie

Spravne atomicky

Atomic<Int> x,y;

Thread 1

X.store(17)

y.store(37)

Thread 2

Cout << y.load() << " ";

Cout << x .load()<< endl;

Chovanie je definovane, moze vystupy 0 0, 0 17, 37 17

Pametovy model v c++11

- Implicitne chovanie zachovava sekvencnu konzistenciu(automaticky vklada odpovedajuce pametove bariery)
- Riziko neefektivneho kodu

Priklad 1

```
atomic<int> x, y;

Thread 1

x.store(17,memory_order_relaxed);

y.store(37,memory_order_relaxed);

Thread 2

cout << y.load(memory_order_relaxed) << " ";

cout << x.load(memory_order_relaxed) << endl;
```

Semantika povoluje v tomto pripade I vystup 37 0

Priklad 2

```
atomic<int> x, y;
```

Thread 1

```
x.store(17,memory_order_release);
y.store(37,memory_order_release);
```

Thread 2

```
cout << y.load(memory_order_acquire) << " ";
cout << x.load(memory_order_acquire) << endl;
```

Acquire nepreusporiada operace load, Release-store

Ine pristupy

Paralelny for cyklus

- Najcastejsi a najjedoduchsia metoda paralelizacie
- Datova paralelizacia

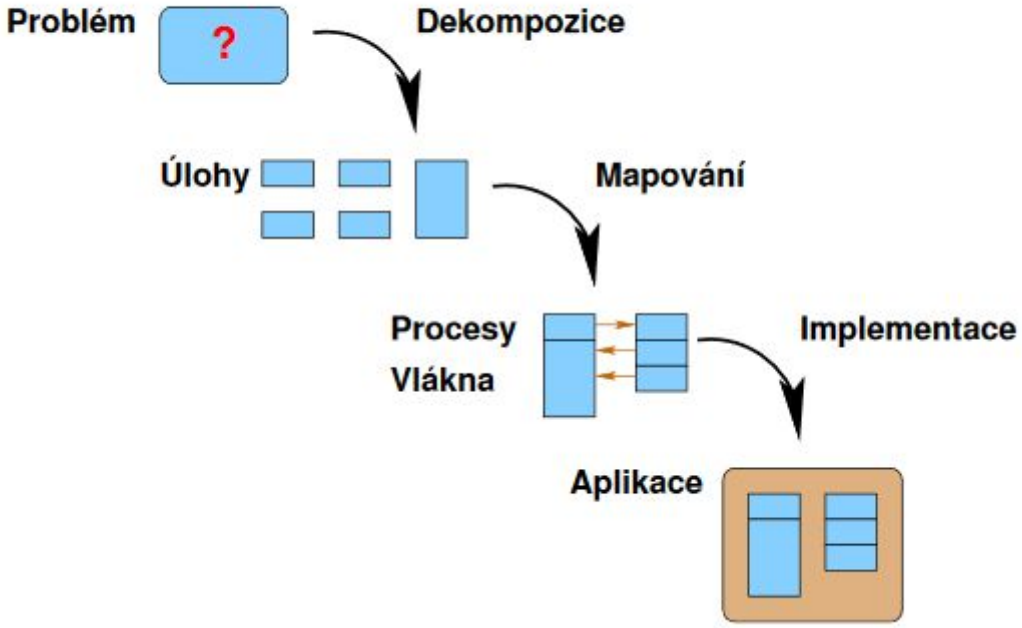
07 – principy navrhu paralelnych algoritmov

Vice-prace programatora paral. Aplikacii

- identifikovat subezne vykonatelne cinnosti a ich zavislosti
- mapovat subezne proveditelne casti prace do procesu
- zaistit distribuciu vstupnych, vnutornych a vystupnych dat
- spravovat subezny pristup k datam a zdielanym prostriedkom
- synchronizovat jednotlive procesy v roznych stadiach vypoctu tak, ako vyzaduje paralelny algoritmus
- mat znalost pridavnych programatorskych prostriedkov suvisiacich s vyvojom paralelnych algoritmov

Zaklady navrhu paralelych algoritmov

Navrh a realizace paralelneho system



Dekompozice a ulohy

Dekompozice

- process rozdelenia celej vypocetnej ulohy na podulohy
- niektore podulohy mozu byt vykonavane paralelne

(pod)ulohy

- jednotky vypoctu ziskane dekompoziciou
- po vycleneni sa povazuju za dalej nedelitelne
- maju uniformnu/neuiniformmnu velkost
- su definovane v dobe kompilace/za behu programu

Priklad

- nasobeni matice A(nxn) vektorom B

Zavislosti uloh

Graf zavislosti

- zachycuje zavislosti vykonavanych uloh
- definuje relativne poradie vykonavanych uloh (ciastocne usporiadanie)

Vlastnosti a vyuzitie grafu

- orientovany acyklicky graf
- graf moze byt nespojity ci dokonca prazdny
- uloha je pripravena k spusteniu, pokial ulohy na ktorych zavisi, dokoncily svoj vypocet(topologicke usporiadanie)

Priklady zavislosti

- poradie obliekania svrsku
- paralelni vyhodnocovani vyrazu
    - v AND x AND (y OR z)

Granularita a stupen subeznosti

Granulaita

- pocet uloh na ktore sa problem dekomponuje
- mnoho malych uloh – jemnozrnna granularita(fine-grained)
- malo vacsich uloh – hrubozrnna granularita(coarse-grained)
- kazdy problem ma vnutornu hranicu granularity

Stupen subeznosti

- maximalny pocet uloh, ktore mozu byt vykonavane sucasne
- limitom je vnutorna hranica granularity

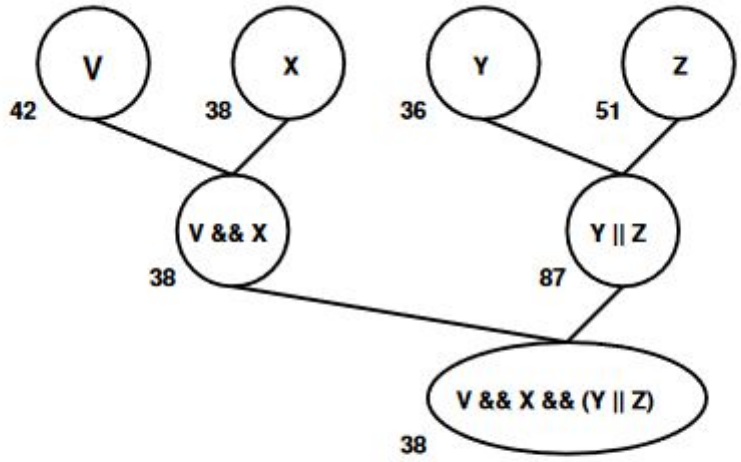
Priemerny stupen subeznosti

- zavisly na grafu zavislosti a granularite
- majme mnozstvo prace asociovane k uzlom grafu
- ***kriticka cesta –*** cesta, na ktorej je sucet praci maximalny
- ***priemerny stupen subeznosti*** - je podiel celkoveho mnozstva prace voci mnozstvu praci na kritickej ceste
- udava maximalne zrychlenie, pokial je cielova platforma schopna vykonavat subezne maximalny stupen subeznosti uloh

Pozorovania

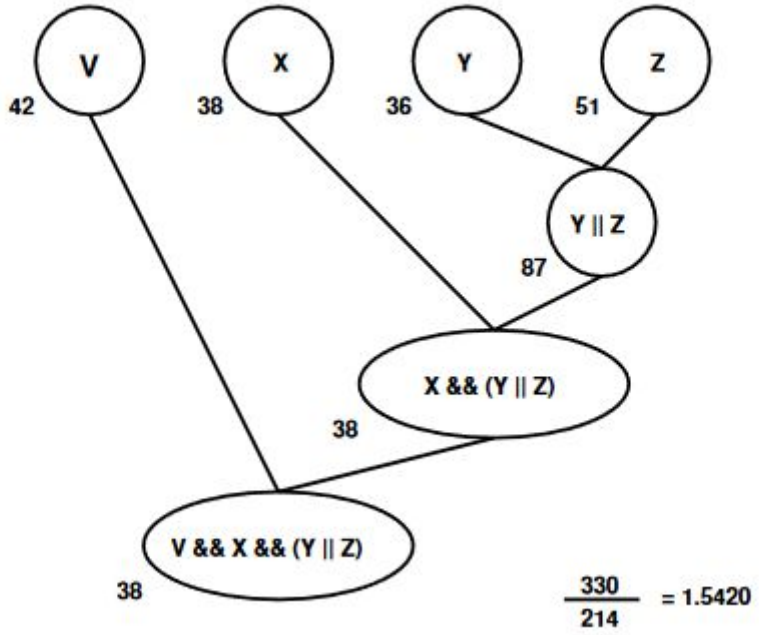
- zjemnovanie dekompozicie moze zvysit stupen subeznosti
- cim menej prace je na kritickej ceste tim vacsi je potencial pre paralelizaciu

Priemerny stupen subeznosti – Varianta 1



$$\frac{330}{176} = 1.8750$$

Priemerny stupen subeznosti – varianta 2



Interakce uloh – dalsi omezeni paralelizace

Interakce uloh

- nezavisle ulohy mozu vzajomne komunikovat
- obojsmerna komunikacia moze znizit stupen subeznosti(ulohy musia co-existovat v stejny okamzik)
- komunikace uloh – ***neorientovany graf interakci***
- graf interakci pokryva graf zavislosti (overeni splneni predpokladu pre spustenie ulohy je forma interakce)

Priklad jednosmernej komunikacie

- nasobenie matice vektorom (y= Ab)
- dekompozice na nezavisle ulohy podla riadku matice A
- prvky vektoru b su citane zo vsekych uloh, je nutne ich vhodne disribuovat k jednotlivym uloham

Techniky dekompozice

Dekompozice

- fundamentalna technika v navrhu paralelnych algoritmov

Klasifikace technik dekompozice

- obecne techniky dekompozice
  - rekurzivni
  - datova
- specializovane techniky dekompozice
  - priezkumova
  - spelukativna

Hybridny pristup

- kombinacia vyssie uvedenych Technik

Rekurzivna dekompozicia

Vhodna pre problem typu rozdel a panuj

Princip

- problem sa dekomponuje na podulohy tak, aby sa jednotlive ulohy mohli mohli dekomponovat rovnakym sposobom ako rodicovska uloha
- niekedy je treba restrukturalizovat ulohu

Priklad

- Quicksort
  - Prevedie sa volba pivota
  - Rozdelenie pola ma prvky mensie nez a vetsi rovno nez
  - Rekurzivne sa opakuje dokial je mnozina prvkov neprazdna
- Hladanie minima v linearnom poli
  - Princip puleni prehladavaneho pola
  - Typicky prikald restrukturalizacie vypoctu

Datova dekompozicia

Zakladny princip

- Data sa rozdelia na casti(data partitioning)
- Ulohy sa vykonavaju subezne nad jednotlivymi castami dat

Datova dekompozicia podla miesta

- Vstupne data
- Vystupne data
- Vnutorne data
- Kombinacia

Mapovanie dat na ulohy

- Funkcie identifikujuce vlakno zodpovedne za zpracovanie dat

Ulohy typu “embarrassingly parallel”

- Trivialna datova dekompozicia na dostatecny poce zcela nezavislych, vzajomne nekomunikujucich uloh

Priezkumova dekompozicia

Princip

- Specializovana technika paralelizacie
- Vhodna pre prehladavajuce ulohy
- Prehliadavany priestor sa rozdeli podla smeru hladania

Vlastnsoti

- Pri znalosti prehliadavaneho stavoveho priestoru sa da dosiahnut optimalneho vyvazenia a zatazenia procesoru
- Na rozdiel od datovej dekompozicie, uloha konci okamzite ked je najdene pozadovane
- ***Mnozstvo vykonanej prace sa lisi od sekvencnej verzie***
- V pripade ze graf neni strom, je treba riesit problem sa opakujucich konfiguracii(riziko nekonecneho vypoctu)

Priklad

- Riesenie hlavolamu “patnact”

Spekulativna dekompozicia

Princip

- Specializovana technika paralelizacie
- Vhodna pre ulohy so sekvenciou datove zavislych poduloh
- Uloha ktora caka na vystup predchadzajucej ulohy sa spusti nad vsetkymi moznymi vystupmi(vystupy predchadzajucej ulohy)

Vlastnosti

- Vykonava sa zbytocna praca
- Nemusi byt vo vysledku rychlejsia ako serializovana verzia
- Vhodna pre ulohy, kde urcita hodnota medzivysledku ma velku pravdepodobnost
- Vznika potencionalny problem pri pristupu k zdrojom(niektore zdroje nemusia byt zdielane v pripade sekvencneho vykonavania uloh)

Priklady

- Spekulativne vykonavanie kodu – vetvenie
- Odhadovanie vysledkov operacii v CPU, zahodenie nespravneho

Zriedka pouzivane

Hybridna dekompozicia

Priklad – hladanie minima v poli

- Sekvencna verzia najde minimum v O(n)
- Pri pouziti datovej a rekurzivnej dekompozicie sa da trvanie tejto ulohy skratit na O(n/p+log(p))
- Vstupne pole sa datovo dekomponuje na p rovnakych casti
- Najdu sa minima jednotlivych casti v case O(n/p)
- Vysledky sa zkombinuju v case O(log(p))
- Teoreticky sa da pri dostatocnom pocte procesorov najst minimum v case O(log(n))

Pozorovanie

- Postup pouzity vo vyssie uvedenom priklade sleduje obecne schema, ktora je oznacovane ako ***MAP-REDUCE***

Techniky mapovania a vyrovnavania zateze

Mapovanie uloh na vlakna/procesy

Mapovanie

- Priradovanie uloh jednotlivym vlaknam/procesom
- Optimalne mapovanie berie v ohlad grafy zavislosti a interakcie
- Ovplyvnuje vykon aplikacie
- Naivne mapovanie (uloha=process/vlakno)
- Je to Mentalny process, tak ako dekompozice

Cile mapovania

- Minimalizovat celkovy cas riesenia celej ulohy
    - redukovat omeskanie sposobene cakanim(idling)
    - redukovat zataz sposobenu interakciou
    - redukovat reziu spustania, ukoncovania a prepinania
    - vyrovnavat zataz na jednotlive procesy
- maximalizovat subeznost
- minimalizovat zatazenia system(zatazenie datovych ciest)
- vyuzivat dostupnost zdrojov pouzitych predchadzajucou ulohou

Charakteristiky uloh, ktore ovplyvnuju mapovanie

Sposob zadania ulohy

- ***staticke zadanie uloh***
    - dekompozicia problem na ulohy je dana v dobe kompilacie, pripadne je priamo odvodena od vstupnych dat
- ***dynamicke zadanie uloh***

    - nove ulohy su vytvarane za behu aplikacie podla priebehu vypoctu pripade ako dosledok vykonavanie povodne zadanych uloh

velkost ulohy

- relativne mnozstvo casu potrebne k dokonceniu ulohy
- uniformna vs neuniformna
- dopredna znalost/neznalost

velkost dat asociovanych k uloze

- snaha o zachovanie locality dat
- rozne data maju roznu rolu a velkost (vstupne/vystpupne data u hlavolamu patnact)

Staticke vs dynamicke

- staticke: prebiehaju v preddefinovanom casovom intervale, medzi predom znamou mnozinou uloh
- dynamicke: pokial predom nepozname pocet interakcii, casovy ramec interakcii alebo participujuce ulohy

Dalsie charakteristiky

- jednosmerna versus obojsmerna interakcia
- mod pristupu k datam -> read-0nly vs read-write
- pravidelne versus nahodile interakce

Rezie suvisiaca s mapovanim do roznych vlakien/procesov

- usposobenie aplikacie pre neocakavanu interakciu
- pripravenost dat k odoslaniu/ adresata k priatiu
- riadenie pristupu k zdielanym zdrojom
- optimalizacia aplikacie pre redukciu prodlev

Schemata pre staticke mapovanie

Mapovanie zalozene na rozdeleni dat

- blokova distribucia
- cyklicka a blokovo-cyklicka distribucia
- nahodna distribucia blokov
- delenie grafu

Mapovanie zalozene na rozdeleni uloh

- delenie podla grafu zavislosti uloh
- hierarchicke delenie

Mapovanie zalozene na rozdeleni dat

Blokova distribucia datovych poli

- procesy spojene s datami rozdelenymi na suvisle bloky
- bloky mozu byt viacrozmerne(redukce interakci)
- priklad
    - nasobenie matic AxB=C
    - delenie matic C na 1- a 2-rozmerne bloky

Cyklicka a blokovo cyklicka distribucia datovych poli

- nervonomerne mnozstvo prace spojene s jednotlivymi prvkami -> blokove delenie sposobuje nerovnomene zatazenie
- blokovo-cyklicka distribuce:delenie na mensie diely a cyklicke priradenie procesom(round robin)
- zmensovanie bloku vedie k cyklickej distribucii(blok je atomicky prvek datoveho pole)

Nahodna distribuce bloku

- zataz suvisiaca s prvkami pola vytvara pravidelne vzory
- -> spatna distribucia v cyklickom rozdeleni
- Nahodne priradenie bloku procesum

Grafove delenie

- Pre pripady, kedy je nevhodne organizovat data do poli(napriklad dratove modely 3d objektov)
- Data organizovane ako graf
- Optimalne delenie
    - Stejny pocet vrcholov v jednotlivch castiach
    - Co mozno najmensi pocet hran medzi jednotlyvimi castmi -> co najmenej komunikacie medzi vlaknami
    - NP uplny problem

Mapovanie zalozene na rozdeleni uloh

Princip

- Graf zavislosti uloh
- Grafove delenie (NP-uplne)

Specialne pripady pre konkretny tvar grafov

- Binarny strom(rekurizna dekompozicia)

Hierarchicke mapovanie

- Ulohove delenie neberie v dotaz neuniformitu uloh
- Shlukovanie uloh do nad-uloh
- Definuje hierarchie(Vrsty)
- Ine mapovacie a dekompozicne techniky na jednotlivych vrstvach

Schemata pre dynamicke mapovanie

Motivacia

- Staticke mapovanie nedostatocne, pretoze charakteristiky uloh niesu zname v dobe prekladu

Centralizovane schemata dynamickeho mapovania

- Ulohy su zhromazdovane v jednom mieste
- Dedikovana uloha pre priradovanie uloh procesom
- Samo-planovanie
    - Akonahle process dokonci ulohu vezme si dalsiu
- Blokove planovanie
    - Pristup k zhromazdistiu uloh moze byt uzkym miestom
    - -> pridelovanie uloh po blokoch

Priklad

- Triedenie prvku v nxn matici A
- For(i=1;i<n;i++) newtask(sort(A[i],n));

Distribuovana schemata

- Mnozina uloh je distribuovana medzi procesmi
- Za behu dochadza k vzajomenmu vymenovaniu uloh
- Netrpi nedostatkami spojenymi s centralizovanym riesenim

Moznosti

- Ako sa urci, kto komu posle ulohu
- Kto a na zaklade coho urci ze je potreba presunut ulohu
- Kolko uloh ma byt presunute
- Kedy a ako je uloha presunuta

Problem

- Efektivita prenosu ulohy na iny process

Afinitne planovanie

Vlakna a procesory

- Jednotlive vlakna su vykonavane fyzickymi procesormi
- Planovanie zaistuje planovac OS

Afinitne planovanie(angl. Affinity scheduling)

- Modifikacie algoritmu planovania
- Afinitne planovanie zaistuje ze vypocetne davky pridelene jednemu procesu/vlaknu budu pokial mozno pridelene na fyzicky tentiez processor

Vyhody a rizika

- Potencionalne lepsie vyuziti cache
- Striktne lpenie na tomtiez procesore moze narusovat vyvazenost vyuzitia procesorov, teda redukovat vykon aplikacie

Dilema procesu dekompozice-mapovani

Otazka

- Je lepsie najprv dekomponovat na mnoho malych uloh a potom ulohy zhlukovat pri mapovani, alebo naopak obmedzit dekompoziciu aby mapovanie bolo priamociare

Aspekty napomahajuce rozhodnutiu dilematu

- Je cena dekompozice zhodna pre oba scenare?
- Vytvara jemnesia dekompozicia skutocne nezavisle ulohy?
- Je jemnejsia dekompozicia zachovana datova lokalita?
- Je/neni znam pocet jader na cielovej platforme?
- Aka je cena rezie prepinani, najma v situaci kedy pocet vlakien vyrazne prevysuje pocet vypocetnich jadier?

Metody pre redukcie rezie interakcie

Rezia suvisiaca s interakciou

- Rezie suvisiaca s interakciou subeznych uloh je klucovym faktorom ovplyvnujucim vykon paralelnej aplikacie
- Z pohladu rezie interakcie su idealne “embarrassingly parallel” ulohy kde k intrakcii nedochadza

Faktory ovplyvnujuce reziu

- Objemy prenasanych dat
- Frekvencia interakcie
- Cena komunikacie

Zvysovanie datovej locality

Ciel – znizit objem prenasanych dat

- Presun zdielanych datovych struktur do lokalnych kopii
- Minimalizacia objemu zdielanych dat
- Lokalizacia vypoctu(lokalne ukladanie medzivysledkov)
- Rezie protokolov pre udrzanie koherencie lokalnych kopii

Ciel – snizit frekvenciu interakcie

- Priestorova lokalizacia prenasanych dat
- Prenasanie dat a ich okolia(princip cache)
- Viac zprav v jednej(bufferovanie)
- Komunikacia castejsie po mensich kusoch alebo komunikacie menej castejsie po vacsich kusoch

Minimalizacia sucasnych pristupov

Problem – contention

- Pristup k omezenemu zdroju v rovnaky okamih(contention) je rieseny serializaciou poziadavok
- Serializacia poziadavkov sposobuje prodlevy

Mozne riesenie

- Je potreba N subeznych pristupov k datam
- Pristupovane data je mozne rozdelit do N blokov
- A data citat v N po sebe iducich iteraciach
- V kazdej iteracii je kazdy blok citan inym vlaknom
- Cislo citaneho bloku v rte iteracii jtym vlaknom (r+j) modulo N

Prekvryvanie vypoctu s interakciou

Problem

- Cakanie na prijem ci odoslanie dat zposobuje nechcene prodlevy

Vcasne vykonanie akcie – podmienky vykonatelnosti

- Data musia byt vcas pripravene
- Prijmajuca I odosielajuca strana mozu asynchronne komunikovat
- Existuje dalsia uloha, ktora moze byt riesena po dobu komunikacie

Ine riesenie

- Simulace mechanizmu prerusenia(ala operacny system)
- Zadne kvoli rezii sposobene nasilnym riesenim

Replikacia dat ci vypoctu

Problem

- Opakovane drahe pristupy k zdielanym datam

Riesenie pre read-only data

- Pri prvotnej interakcii tvorba kopii dat(datova lokalita)
- Dalej pracovat s lokalnou kopiou
- Zvysuje pametove naroky vypoctu

Riesenie pre read-write data

- Podobne ako v read-only pripade
- Nasobne subezne vypocty toho isteho mozu byt rychlejseie nez citanie a zapis zdielanej hodnoty

Optimalizovane operacie pre kolektivnu komunikaciu

Problem

- Stejna interakcia medzi vsetkymi procesmi vykonava zakladnymi komunikacnimi primitivy je draha

Riesenie – kolektivni komunikacne operacie

- Pre pristup k datam inych vlakien/procesov
- Dolezite pre komunikacne intenzivne vypocty
- Forma efektivnej synchronizacie

Optimalizovane implementacie

- MPI

Prekryvanie interakcii

Problem

- Nedostatocna priepustnost komunikacnej siete, ci absencie kolektivnych komunikacnych operacii

Riesenie

- Zvysit vyuzite komunikacnej siete sucasnou komunikaciou medzi roznymi parmi procesov

Priklad

- 4 procesy P1,⋯P4
- P1 chce vsetkym poslat spravu m1
- P1->P2,P2->P3,P3->P4
- P1->P2, (P2->P3, P1->P4) -> poslu sa dve spravy naraz z P2 a P1

Redukcia ceny komunikacie

Komunikacia v nezdielanom adresovom priestore

- Syncrhonizacia posielanim sprav

- Predavanie dat posielanim zprav

Komunikacia v zdielanom adresovom priestore

- Synchronizacia korektnym pristupom k zdielanym datam
- Predavanie dat pomocou zdielanych datovych struktur(FIFO)

Obecne charakteristiky

- ***Latence*** -> doba potrebna pre dorucenie prveho bitu
- ***Prenosova rychlost*** – objem dat prenesenych za jednotku cas

Latence

- Celkova cena pre zahajenie komunikacie – ts
  - Cas pre pripravu zpravy/dat
  - Identifikacia adresata/routovanie
  - Doba trvania vyliatia informacie z cache do pameti pripadne na sietove rozhranie
- Cena "hopov"(preposielani vo vnutri komunikacnej siete) – th
  - Cas straveny na jednotlivych routeroch v sieti
  - Doba po ktoru putuje hlavicka zpravy z prijmaceho na odosielaci port

Prenosova rychlost

- Ovplyvnene sirkou pasma r
- Cena za prenos jedneho slova(Word=2bajty) -> tw = 1/r

Cena komunikacie

- M -> delka zpravy v slovach
- L – pocet liniek cez ktore zprava putuje
- Ts + l * (th + mtw)

Obecne metody redukcie ceny

- Spojovanie malych sprav(amortizuje sa hodnota ts)
- Kompresia(znizovanie hodnoty m)
- Minimalizacia vzdialenosti(znizovanie hodnoty l)
- Paketovanie(eliminacia rezie zposobenej jednotlyvimi hopmi)
  - Ts + l*(th+mtw) -> ts + lth + mtw

08- kolektivne komunikacne primitiva

Kvantitativne parameter komunikacie

Komunikace(interakce)

- Predavanie informacii medzi jednotlive procesy.

Parametry komunikace

- ***Latence*** -> oneskorenie suvisiace so zapocatim vlastnej komunikacie
- ***Sirka pasma(bandwith)*** -> maximalne mnozstvo dat prenesenych za jednotku casu
- ***Objem*** -> mnnozstvo predavanych dat

Cena komunikacie

- Ts -> latence, tw-> sirka pasma, m -> objem
- Ts + m*tw = cena

Priklady

- Za ako dlho napustim hrncek vodou pomocou 2km dlhej zahradnej hadice
    - Analogia pre vysoku latencie, aj ked objem dat je maly
- Pri konstantnom citani z pamete trva ziskanie kodu instrukcie a prislusnych operandov z pamete v priemere 5ns, aka je najvyssia realna rychlost vykonavania kodu ulozeneho v pameti 4GHz procesorom?
    - [1/(5*10^-9)=0.2*10^9=200 MHz]

Efektivita interakce

Pozorovani

- Interakcia jednotlivych uloh/procesov je nevyhnutelna
- Interakcia zposobuje prodlevy vo vypocte
- Rezie suvisiaca s interakciou by nemala byt dovodom pre neefektivitu paralelneho zpracovania

Zaver

- Komunikacne primitive musia byt co najefektivnejsi

Topologie komunikacnej siete

Topologie

- Fyzicka/logicka struktura komunikacnych kanalov medzi jednotlivymi participantnmi komunikacie

Vlastnosti komunikacnej siete

- Priemer (dlzka maximalnej najkratsej cesty)
- Konektivita (minimalny pocet disjunktnych ciest) -> robustnost
- Stupen (max pocet liniek vychadza z jedneho uzlu
- Cena
- Vylucnost pristupu (pouzitie jednou ulohou blokuje ostatne)
- Rozsiritelnost

- Skalovatelnost

Sbernice

- Komunikacia cez jedno zdielanie miesto/ zdielane medium
- Priepustnost, klesa s poctom uzlov(je treba cache)
- Dolezitost: model zdielaneho adresoveho priestoru

Kliky(uplne siete)

- Private neblokujuce spojenia kazdeho s kazdym
- Cena: pocet liniek je kvadraticky voci N
- Cena: stupen kazdeho uzlu je N-1
- Skalovatelne, za podmienky levne zvysovania stupne uzlu
- Dolezitost: abstraktna predstava siete, logicka struktrua
- Najpouzivanejsi model

Prsten(kruh, retaz, 1-rozmerna mriezka)

- Usporiadanie na uzloch
- Privatna neblokujuca komunikacia s 2 najblizsimi uzlami
- Dolezitost: logicka struktura komunikacie, pipeline model

Hviezdice

- Spjenie cez jeden centralny uzel
- Stredovy uzel moze by uzkym miestom
- Lze hierarchicky vrstvit (hviezdica hviezdic)
- Dolezitost: master-slave model

Hyperkostka

- Spojenie n uzlov v tvae log2n-rozmernej krychle
- Stupen uzlu je log2n
- Priemer siete je log2n
- Pocet liniek O(N.log(N))
- Postup konstrukcie
    - Binarne oznacenie uzlov s identifikatormi 0 az 2^(log2n) -1
    - Linka existuje medzi uzly pokial sa oznacenie lisi v jednom bite
    - Minimalna vzdialenost uzlov odpoveda poctu odlisnych bitov v oznaceni uzlov

Strom s aktivnymi vnutornymi uzly

- Strom, kde participantanmi komunikacie su listy I vnutorne uzly
- Kostra hyperkostky – strom s maximalnou hlbkou log2n
- Dolezitost: optimalne sirenie informacii

Ine topologie -> velmi specificke pripady

- Mriezky

- Cyklicke mriezky(torrus)

Jeden na vsechny a vsichni na jedneho

One-To-All vysielanie(OTA)

- Uloha posiela niekolko/vsetkym ostatnym identicka data
- Vo vysledku je *p* kopii originalnej zpravy v lokalnych adresovych prostoroch adresatu
- Zmena struktury dat m-> p*m (m-je velkost spravy)

All-To-One redukcia (ATO)

- Dualna operacia k “One-To-All”
- Niekolko/vsetky ulohy posielaju data jednej ulohe
- Data sa kombinuju pomocou zvoleneho asociativneho operatoru
- Vo vysledku je jedna kopia v adresovom priestore cielovej ulohy
- $M1 \otimes m2 \otimes m3 ... \otimes mp \rightarrow m$
- Zmena struktury dat m*p -> m
- Dochadza k redukcii

OTA: Pre topologie prsten ci retaz

Naivny sposob One-To-All pre p procesov

- Poslat p-1 zprav postupne ostatnym procesom
- Uzke miesto: odosielatel
- Siet je nevyuzita, komunikuje iba jedna dvojica pocesov

Technika rekurzivneho zdvojenia(pripomenutie)

- Nejprve prvni process posle zpravu inemu procesu
- Potom oba procesy poslu zpravu inej dvojici
- Potm ctverice procesov posle zpravu inej stvorici
- Atd
- Prvy process posle najviac log(p) zprav
- Subeh zprav na linkach siete
- Optimalna vzdialenost adresatov pre jednotlive kola su p/2, p/4, p/8, p/16,$\cdots$

OTA: Pre topologiu d-dimenzionalna mriezka

One-to-All v 2-rozmernej sieti o p uzloch

- Lze chapat ako sqrt(p) retizku o srt(p) uzloch
- V prvej fazi sa propaguje informacia do vsetkych retizku
- V druhej fazi sa zubezne propaguej informacia v jednotlivych retiazkoch
- Celkova cena 2(log(sqrt(p)))

One-To-All v d-rozmernej sieti

- Stejny princip, velkost v jednom rozmere je p^(1/d)

- Celkova cena d(log(p^(1/d))))

OTA pre topologiu hyperkostka

Hyperkostka s 2^d vrcholmi

- Aplikuje sa na algoritmus pre siet vo tvare mriezky
- D-dimenzionalna siet s hlbkou siete 2
- Kazda faze prebehne v konstantom case(poslane 1 spravy)
- Nenastava subezny pristup na ziadnu z komunikacnych liniek
- Celkova cena: d

Priklad

- Jak probehne OneTo-All na hyperkostce s dimenzi 5

ATO: dualne k OTA

All-To-One redukce pre p procesov

- Prebieha dualne k operacii one-to-all

Priklad

- Prve procesy s lichym id poslu zpravu procesom s ID o jedna mensim, kde sa zpravy zkompinuju s hodnotou, kterou chteji vyslat procesy so sudym ID
- Nasledne probehne kombinacia informacii susednych procesov so sudym ID na procesoch jejichz ID je nasobkem ctyr
- Atd

Univerzalne algoritmy

Pozorovanie

- One to all procedury su si podobne
- Jdu nahradit univerzalnou procedurou

Univerzalne algoritmy OTA vysielanie a ATO redukce

- Predpokladaju 2^d uzlu(hyperkostka)
- Kazdy uzel identifikovan bitovym vektorem
- AND a XOR bitove operace

Cena

- D(ts+mtw)
- D pocet dimezi hyperkostky

*

Vsetci vsetkym a od vsetkych vsetkym

All-to-All vysielanie

- Vsetci posielaju informaciu vsetkym
- Kazda uloha posiela jednu zpravu vsetkym ostatnym uloham
- Vo vysledku je p kopii p originalnych zprav v lokalnych adresovych prostoroch adresatu
- Zmena struktury dat p*m -> p*p*m

All-to-All Redukce

- Vsetci posielaju informaciu vsetkym, prichozi spravy sa kombinuju
- Kazda uloha ma pre kazdu inu ulohu inu zpravu
- Zmena struktury dat p*p*m -> p*m

Naivne riesenie

- Sekvencne/nasobne pouzitie One-To-All procedur
- Neefektivne vyuzitie siete

ATA: pre topoogiu prsten ci retaz

Prsten

- 1. Faze kazdy uzel posle informaciu svojmu susedovi
- 2 az nta faze: uzly zbieraju a preposielaju prichadzajuce spravy
- Vsetky linky su po celu dobu operacie plne vyuzite
- Optimalny algoritmus

Retez

- Vysielane sprav susedom na obe strany
- Full-duplex linky -> n-1 fazi
- Half-duplex linky -> 2(n-1) fazi

ATA redukce

- Zpravy po jednej posielane po kruhu tak, ze zprava pre najvzdialenejsi uzel je poslana ako prva
- Pri priechode uzlom sa prikombinuje lokalna zprava pre adresata prave prechadzajucej spravy

ATA: pre topologiu mriezky ci hyperkostky

Mriezka

- 2 faze – sqrt(p) retiazkov o sqrt(p) uzloch
- Po prvej fazi uzly maju uzly sqrt(p) casti z celkoveho poctu p casti zpravy
- Priklad siete 4x4, kazda posiela svoje ID kazdemu

Hyperkostka

- Rozsirenie algoritmu pre siete na d dimenzii
- Po kazdej fazi (z celkoveho poctu d j)e objem zprav zdvojnasobeny

ATA redukcia

- Dualny postup
- Po kazdej fazy je objem zprav zredukovany na polovicu

ATA: analyza ceny

Kruh a linearne pole, p uzlov

- $T= (t_s+t_w m)(p-1)$

2d mriezka, p uzlov

- 1. fáze$(t_s+mt_w)(\sqrt{p}-1)$
- 2. fáze$(t_s+m\sqrt{p}t_w)(\sqrt{p}-1)$
- Celkem:$T=2t_s(\sqrt{p}-1) +t_w m(p-1)$

Hyperkostka, p=2^d uzlov

- $T=\sum_{i=1}^{\log p}(t_s+2^{i-1}t_w m)$
- $T=t_s \log p+t_w m(p-1)$

Pozorovanie

- clen $t_w m(p-1)$ se vyskytuje vždy
- pipeline niekolko OTA operacii

Vsichni vsem invividualni komunikacia

All-to-All invividualni komunikace (ATA invividualni)

- kazda uloha posiela rozne data ostatnym uloham
- dojde k vymene 2D pola zprav(pxp) v 1D priestore(p)
- take oznacovane ako totalna vymena
- zmena struktury dat:
  - $p*(m_1, \ldots ,m_p) \rightarrow p*m_1, p*m_2, \ldots , p*m_p$

Priklad

- transpozie matice (A^T[i.j] = A[j,i])
- matice nxn mapovana po riadkoch na n uloh
- uloha j ma k dispozici prvky [j,0],[j,1],⋯[j,n-1]

ATA inviviualni: pre topologiu prsten ci retez

Princip

- najprv ulohy poslu jednym smerom zpravy pre zbyvajucich p-1 uloh
- v kazdom dalsom kole kazda uloha vyextrahuje zpravu ktora je urcena jej a zbyvajuce spravy preposle
- v poslednom kole vsechny ulohy preposielaju len jednu spravu

Analyza ceny

- $T=\sum_{i=1}^{p-1}(t_s+t_w m(p-i))=t_s(p-1)+\sum_{i=1}^{p-1} i t_w m = (t_s+t_w mp/2)(p-1)$

ATA inviviualni pre topologiu mriezky

Princip

- Siet je sqrt(p) retizku o delce sqrt(p)
- 1. Faze: medzi retizky sa vymenia v jendom smere zpravy tak, aby sa informacia pre kazdy uzol v retazku bola niekde retiazku obsiahnuta
- 2. Faze v ramci retizku sa informacia napropaguje na odpovedajuce miesto

Priklad

- ATA invividualni komunikace na sieti 4x4 uzly

Cena

- Každá fáze distribuuje zprávy velikosti $m\sqrt{p}$ mezi $\sqrt{p}$ uzly
- Cena jedné fáze (viz cena pro prsten):
  - $(t_s+t_w mp/2)(\sqrt{p}-1)$
- $T= (2t_s+t_w mp)(\sqrt{p}-1)$

ATA inviviualni pre topologiu hyperkostky

Princip

- Aplikacia algoritmu pre d-dimenzionalne siete
- Podel jedne dimenzie sa posiela vzdy p/2 sprav

Priklad

- Krychle 2x2xx2 uzlu

Cena

- V kazdej faze vymenene mp/2 dat
- Log p fazi
  - $T= (t_s+t_w mp/2)\log p$
- Naviac v kazde faze uzly preusporiadavaju/tridia spravy

ATA inviviualni pre topologiu hyperkostky – optimalita

Problem

- **Kazdy uzol posiela a prijma m(p-1) dat**
- Průměrná vzdálenost, na kterou data putují je(log p)/2

- Celkový provoz na síti je $p \times m(p-1) \times (\log p)/2$
- Počet linek v hyperkostce je $p(\log p)/2$
- Optimální algoritmus by měl dosáhnout složitost
    - $T = t_w pm(p-1)(\log p)/2(p \log p)/2 = t_w m(p-1)$

Zaver

- Pre ATA invividualni komunikaci neni algoritmus pre siete vo tvare mrizi optimalni na sietoch v tvare hyperkostky

*

E-cube routing -> IB109 -> video31 -> este raz pozriet

E-cube routing

- Cesta medzi 2 bodmi je dana poziciou odlisnych bitov tychto bodov, routuje sa od najmenej vyznamneho bytu
- Vyzaduje duplexne linky

Operace kompletnej redukcie a prefixoveho suctu

Operace kompletej reducie

Kompletni redukce – All reduce

- Ulohy si vzajomne vymenuju data ktora sa kombinuji
- Lze realizovat jako ATO redukci nasledovanou OTA vysielanim vysledku z predchoczej redukce
- Zmena struktury dat p*m -> p*m

Semtantika a vyuzite pre ucely synchronizacie

- Uloha nemoze dokoncit operaciu, dokial vsetky participujuci ulohy neprispeviaju svojim dielom
- Moze byt pouzite pre realizaciu synchronizacneho primitiva

Implementace

- Naivne pomocou ATO a OTA nebo
- Modifikacia operacie ATA vysielana
    - Rozdiel od ATA – spravy sa nekumuluju ale kombinuju
- Cena pro hyperkostku je $T = (t_s + t_w m) \log p$

Operace prefixoveho suctu

Prefixov sucet

- Ulohy posielaju data ostatnym uloham so stejnym ci mensim ID
- Data sa kombinuju – redukcia
- Zmena struktury dat p*m -> p*m

Priklad

- Data v jednotlivych uzloch <3,1,4,0,2>
- Kombinacia pomocou operace suctu
- Vysledna data <3,4,8,8,10>

Implementacia

- Pouzitie algoritme pre ATA
- Ako sa pozna ze zprava pochadza od uzlu z mensim ID?
- Vsetke posielana data obohatena o ID puvodce

Operace Scatter and Gather

Scatter – tez oznacovane ako "Scan"

- Jedna uloha posiela unikatnu zpravu kazdej dalsej ulohe
- One to all invividualni(personalized) komunikace
- Na rozdiel od OTA sa Ziadne data sa neduplikuju
- Zmena struktury dat (m1,⋯mp) -> m1, ⋯,mp

Gather – tez oznacovana ako "Concatenation"

- Jedna uloha zbiera unikatne data od ostatnych uloh
- All to one invividualni (personalized) komunikace)
- Na rozdiel od ATO redukcie sa nevyskytuje kombinacia dat
- Zmena struktury dat m1,⋯,mp -> (m1, ⋯,mp)

Implementace

- Vuzitie algoritmu pre ATO a OTA
- Celkem log p fazi, s kazdou fazou sa objem dat zvacsuje
- $T=t_s(\log p) + t_w m(p-1)$

Operace cyklicky posun

Permutace

Permutace

- Obecnejsi komunikacni primitivum
- Sucasne priebehajuci OTO prerozdelovania dat
- Jedna uloha posiela data jednej inej ulohe

Priklad

- Cyklicky posun o q(circular q-shift)
- P uloh
- Uloha I posiela data ulohe (i+q) mod p

Pouzitie

- Specificke maticove operacie
- Vyhladavanie vzoru v texte ci obraze

Cyklicky posun

1-dimenzionalni

- Intuitivne -> rotace o q pozic
- Smer rotace zavisly na q, urci sad lde vyrazu min (q,p-q)

2d siet

- Akcelerace cyklickeho posunu s vyuzitim druhe dimenze nez ve ktere probiha posun
- Jedna dimenze ma rozmer sqrt(p) uzlu
- Posun v druhej dimenzi akceleruje o sqrt(p) kroku

Cena

- Nejvzdalenejsi posun v jednej dimenzi je sqrt(p)/2
- $T= (t_s+t_w m)(\sqrt{p})$

Priklad

- Siet 4x4 uzly, cyklicky posun o 5

Cyklicky posun – siet v tvare hyperkostky

Princip

- Mapovanie linearneho pola na hyperkostku dimenze d
  - Zrcadleny sedy kod(reflected grey code): I = G(I,d)
  - G(0,1) = 0
  - G(1,1) = 1
  - $G(i,x+1) =ifi<2^x G(i,x)else\ 2^x+G(2^{x+1}-1-i,x)$
- Q sa vyjadri ako sucet mocnin cisla 2
- Posun prebehne v tolko fazach kolko je clenov v sucte

Cena

- Uzly v vzdialenosti mocniny cisla 2, su od seba vzdialene najviac 2 kroky(vlastnost kodu)
- Clen v sucte je najviac log p
- Komunikacia prebieha bez blokovania liniek
- $T= (t_s+mt_w)(2\log p)$
- Se zpětným posunem je počet sčítanců max.(log p)/2
- $T= (t_s+mt_w)(\log p)$

09 – programovanie aplikacii pre prostredie s distribuovanou pametou

Principy programovania s predavanim sprav

Vymezeni prostredi

Paradigma – predavanie sprav

- Nezdielany adresovy priestor
- Explicitny paralelizmus

Pozorovanie

- Data explicitne delena a umiestnena do jednotlivych lokalnych adresoych priestorov
- Datova lokalita je klucova vlasnost pre vykon
- Komunikacia vyzaduje aktivnu ucast komunikujucich stran
- Existuju efektivne implementovana podporne knihovne
- Programator je zodpovedny za paralelizaciu algoritmu

Stuktura programov predavanim sprav

Asynchronni paradigma

- Vypocet zacne v rovnaky okamzik(synchronne) ale
- Prebieha asynchronne(Rozne vlakna roznou rychlostou)
- Mozna synchronizacia v jednotlivych bodoch vypoctu
- Neplati “trojuholnikova nerovnost” v komunikacii
    - Moze sa stat ze ked Vlakno A posiela vlaknam B a C spravy, a vlakno B posiela tu istu spravu C, tak moze sprava od B do C dorazit skor nez sprava od A do C

Dalsie vlastnosti

- Vykazuje nedeterministicke chovanie
- Tazsie prekazanie korektnosti
- Moznost vykonana zcela odlisneho kodu na jednotlivych procesnych jednotkach
- Typicky vsak “single program multiple data”
- Kazda procesna jednotka ma jednoznacu identifikaciu

Send a Receive – zakladne stavebne kamene

```
send(void *sendbuf, int nelems, int dest)
```

`receive(void *recvbuf, int nelemns, int src)`

sendbuf

- ukazatel na buffer(blok pameti), kde su umiestnene data pripravene k odoslaniu

recvbuf

- ukazatel na bafr(blok pameti), kam budu umiestnene priate data

nelems

- pocet datovych jednotiek, ktore budu poslane ci priate(delka zpravy)

dest

– adresat odosielanej zpravy, tj ID toho, komu je zprava urcena

src

- odosielatel spravy, ID toho kto spravu poslal, nebo toho od koho chcem spravu prijat

*

Vystup a efektivita

- asynchronna semantika send() a receive()
  - inac sa receive() vola blokujuco
- nutne z dovodu zachovania vykonu aplikacie

Blokujuce nebafrovane operacie

Blokujuce operacie

- operacia send() ukoncena az tedy, jelito bezpecne vzhladom k semantike, tj ze prijemce obrdzi to co bylo obsahem odosielaneho bafru v okamziku volania operacie send()
- ukoncenie operacie send() nevynucuje a negarantuje ze prijemce jiz zpravu prijal
- operace receive() skonci po priati dat a jejich umiestnenia na spravne miesto v pameti

Nebafrovane operace

- neni medzipamet kde by sa docasne ulozil obsah toho co sa ma odoslat
- operace send() skonci az po dokonceni operace komunikace, tj az prijmaci process zpravu prijme
- v ramci operacii send() a receie() pred samotnym prenosom dat prebieha synchronizacia oboh participujucih stran(handshake)
- nebafrovane operacie davaju najvacsi zmysel pri posielani velkych dat
- pre male spravy dava zmysel mat bufrovane spravy

Blokujuce nebafrovane operacie - problemy

Prodlevy

- sposobene synchronizaciu pred vlastnou komunikaciou
- process ktory dosiahne bodu, kedy je pripraveny komunikovat, caka az do stejneho bodu dospeje I druhy process
- volanie send() a receive() v rovnaky okamzik nelze garantovat na urovni kodu
- nema vliv, pokial dominuje cas komunikacie

Deadlock

- send a receive mozu vyvolat deadlock ak sa nevola odpovedajuci receive

*

Blokujuce bafrovane operacie

Bafr v bafrovane komunikacii

- extra pamet zdanlivo mimo adresovy priestor procesov
- medzisklad zprav pri komunikacii

bafrovane komunikacne operacie

- operacia send() skonci v okamihu, kedy odosielane data kompletne prekopirovane do bafru
- pripadna modifikacia posielanch dat po skonceni operacie send(), ale pred zapocatim vlastnej komunikacie sa neprojevi
- volanie send(), vlastna komunikacia a nasledne volani receive() na prijmacej strane sa nemusi casovo prekryvat

Blokujuce bafrovane operacie- problemy

Rezia suvisiaca s bafrovanim

- eliminacia prodlev za cenu rezie bafrovani
- vo vysoko synchroniich aplikaciach moze byt horsie nez pouzivane blokujucich nebafrovanych operacii

velkost bafru

- pokial odosielatel generuje zpravy rychlejsie, nez je prijemca schopny spravy prijmat, velkost bafru moze neumerne rast(problem producent-konzument)
- pokial je velkost pre bafry obmedzena, moze dochadzet(a to nedeterministicky) k situacii, kedy je predom dana velkost bafru nedostatecna(buffer overflow)
    - bud sa spravy zahadzuju, alebo sa volanie send() stane nebafrovane po nejakej dobe a musi cakat az prijemca zpravy z bufru zkonzumuje
- pripadne samovolne blokovanie odosielatela do tejto doby nez odosielatel prijme nejake data a bafry sa uvolnia, muze vest k uviaznutiu, podobne ako v pripade nebafrovanej blokujucej komunikacie

Uviaznutie (I bez blokovania odosielatela)

- ak najprv zavolam receive a potom send, dostanem deadlock *

asymetricky model blokujuci bafrovane komunikacie

- neexistenca odpovedajucich prostriedkov pre bafrovanu komunikaciu na urovni komunikacnej vrstvy
- operacia send() je blokujuca, nebafrovana
- prijmaci process je preruseny v behu a zprava je prijata do bafru, kde caka dokial prijmaci process nezavola odpovedajucu operaciu receive()
- dedikovane vlakno pre obsluhu komunikacie

neblokujuce operacie

Motivacie

- komunikacia, ktora nesposobuje prodlevy

neblokujuce operacie

- volanie funkcie moze skoncit drive nez je to semanticky bezpecne
- existuje funkce na zjisteni stavu komunikujucej operacie
- program nesmie modifikovat odosielane data, dokial komunikacia neskonci
- po dobu trvania komunikacie program moze vykonavat kod
- prekryvanie vypoctu a komunikacie

hw realizacia

- DMA

Prehled komunikacnych modov

	Blokující	Neblokující	
Bafrované	send skončí jakmile jsou data nakopírována do bafru	send skončí jakmile je inicializován DMA přenos	Asynchronní obsluha bufru, vyšší paměťové nároky.
Nebafrované	send skončí až po skončení odpovídajícího receive	send skončí ihned, odešle se pouze požadavek na komunikaci	Nutnost časové synchronizace účastníků komunikace.
	Sémantiku operací nelze porušit	Korektní dokončení operací nutné zjišťovat opakovaným dotazováním se	

Message Passing Interface

Message Passing Interface

Standard MPI

- Standardizuje syntax a semantiku komunikacnych primitiv
- Rozhranie pre C, Fortran
- MPI verze 1.2
- MPI verze 2.0(paralelne I/O, c++ rozhrani)
- Existuju rozne implementacie standard
    - Mpich
    - LAM/MPI
    - Open MPI

Minimalne pouzitie MPI

MPI funkce	Význam
MPI_Init	Inicializuje MPI
MPI_Finalize	Ukončuje MPI
MPI_Comm_size	Vrací počet participujících procesů
MPI_Comm_rank	Vrací identifikátor volajícího procesu
MPI_Send	Posílá zprávu
MPI_Recv	Přijímá zprávu

Definicia typu a constant: *#include "mpi.h"*

Navratova hodntota pri uspesnom volani funkcie *MPI_SUCCESS*

Kombiplace: *mpicc, mpiCC, mpiC++*

Spustenie programu: *mpirun*

Dobre prostredie pre MPI su zdielane domovske adresare, cez viacero strojov, mat dobre nastavene ssh kluce(nastavenie prihlasovanie sa na stroje bez hesiel)

MPI komunikatory

Komunikacne domeny

- Zdruzovanie participujucich proesorov do skupin
- Skupiny sa mozu prekryat

Komunikatory

- Premenne ktore uchovavaju komunikacne domeny
- Typ *MPI_Comm*
- Su argumentum vsetkych komunikacnych funkcii MPI
- Defaultny komunikator: *MPI_COMM_WORLD*
    - Komunikator do ktoreho su pripojene vsetky participujuce procesy, taka ko boli pri zapnuti aplikacie

Zistenie velkosti domeny a identifikatoru vramci domeny

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
    - `kolko clenov ma skupina daneho komunikatoru`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
    - `zistim rank vramci komunikatoru`
- `rank je identifikator procesu v danej domene`
- `rank cislo v intervale [0,size-1]`

*

Posielanie zprav

```
int MPI_Send(void *buf, int count,MPI_Datatype
datatype, int dest,int tag, MPI_Comm comm)
```

- odosle data odkazovana ukazatelom *buf*
- na data sa pozera ako na sekvenciu instancii typu *datatype*
- odosle sa *count* po sobe iducich instancii
- *dest* je rank adresata v komunikacnej domene urcenej komunikatorom *comm*
- *tag*
    - prilozena informacia typu *int* v intervale [0, MPI_TAG_UB]
    - pre prijemce viditelna bez citania obsahu zpravy

- typicky odlisuje typ zpravy

Korespondence datovych typu MPI a C

MPI datový typ	Odpovídající datový typ v C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

MPI ma vlatne typy, lebo umoznuje bezat na roznych typoch procesoroch parallelne -> ine usporiadanie bytov, 32bvs64b

Prijmanie zprav

```
int MPI_Recv(void *buf, int count,MPI_Datatype
datatype, int source,int tag, MPI_Comm comm,MPI_Status
*status)
```

- prijmeme zpravu od odosielatela s rankom *source* v komunikacnej domene *comm* s tagom *tag*
- source moze byt *MPI_ANY_SOURCE*
  - *je mi jedno kto to poslal*
- tag moze byt *MPI_ANY_TAG*
- Zprava ulozena na adresa urcene ukazatelom *buf*
- Velkost bafru je urcena hodnotami *datatype* a *count*
- Pokial je bafr maly, navratova hodnota bude *MPI_ERR_TRUNCATE*

Prijmanie zprav – MPI_Status

```
MPI_Status
```

```
typedef struct MPI_Status {
```

```
int MPI_SOURCE;

int MPI_TAG;

int MPI_ERROR;

};
```

- Vhodne najma v pripade prijmu v rezime *MPI_ANY_SOURCE* nebo *MPI_ANY_TAG*
- *MPI_Status* drzi I dalsie informacie, napriklad skutocny pocet prijadych dat(delka zpravy)

```
int MPI_Get_count(MPI_Status *status,MPI_Datatype datatype,int *count)
```

Posielanie a Prijmanie zprav – semantika

MPI_Recv

- Volanie skoncia az su data umiestnene v bafru
- Blokujuci receive operace

MPI_Send

- MPI standard pripusta 2 rozne zemantiky
  - Volanie skonci az po dokonceni odpovedajucej receive oprace
  - Volanie skonci jakmile su posielane data zkopirovane z bufru
- zmena odosielanych dat je vzdy semanticky bezpecna
- blokujica send operace

Mozne dovody uviaznutia

- jine poriade zprav pri odosielani a prijmani
- cyklicke posielanie a prijmanie zprav(vecerajuci filozofove -> cyklicky si do kruhu posielaju spravu)

Sucasne Posielanie a Prijmanie zprav

MPI_Sendrecv

- operacia pre sucasne prijmanie a odosielanie zprav
- nenastava cyklicke uviaznutie
- bafry pre odosielane a prijmane data musia byt rozne

```
int MPI_Sendrecv (void *sendbuf, int sendcount,MPI_Datatype senddatatype, int dest, int sendtag,void *recvbuf, int recvcount,MPI_Datatype
```

```
recvdatatype,int source, int recvtag, MPI_Comm
comm,MPI_Status *status)
```

Sucasne Posielanie a Prijmanie zprav – Sendrecv_replace

Problemy *MPI_Sendrecv*

- bafry pre odosielana a prijmane data zabieraju 2x tolko miesta
- zlozita manipulacia s datami vdaka 2 roznym bafrom

MPI_Sendrecv_replace

- operace odosle data z bafru a na jejich misto nakopiruje prijata data
- len syntakticky cukor, ono je to stale rozdelene na 2 miesta

```
int MPI_Sendrecv_replace (void *buf, int
count,MPI_Datatype datatype, int dest, int sendtag,int
source, int recvtag, MPI_Comm comm,MPI_Status *status)
```

Neblokujuca komunikacia

Neblokující Send a Receive

```
int MPI_Isend(void *buf, int count,MPI_Datatype
datatype, int dest,int tag, MPI_Comm comm,MPI_Request
*request)
```

```
int MPI_Irecv(void *buf, int count,MPI_Datatype
datatype, int source,int tag, MPI_Comm comm,MPI_Request
*request)
```

- velke I na zaciatku hovori ze sa jedna o neblokujucu funkciu

MPI_Request

- identifikator neblokujucej komunikacnej operacie
- potreba pri dotazovani sa na dokoncenie operacie

Dokoncenie neblokujucich komunikacnych operacii

int MPI_Test(MPI_Request *request, int *flag,MPI_Status *status)

- Nulový flag znamená, že operace ještě probíhá
  - 0 prebieha, nenula nepriebieha/bola dokoncena

- Při prvním volání po dokončení operace se naplní status, zničí request a flag nastaví nature

int MPI_Wait(MPI_Request *request,MPI_Status *status)

- Blokující čekání na dokončení operace
    - Urobi z neblokujucej operacie blokujucu
- Po dokončení je request zničen a status naplněn

int MPI_Request_free(MPI_Request *request)

- Explicitní zničení objektu request
- Nemá vliv na probíhající operaci

Neblokujuce operacie – poznamky

Parovanie

- Neblokujuce a blokujuci send a receive sa mozu lubovolne parovat

Uviaznutie

- Neblokujuce operace riesia vacsinu problem s uviaznutim
- Neblokujuce operacie maju vzdy vyssie pametove naroky
- Pozdejsie zahajena neblokujuca operacia moze skoncit drive

Kolektivni komunikace

Kolektivni operace

- Mnozina participujucich procesov je urcena komunikacnych domenou(MPI_Comm)
- ***Vsetky procesy v domene musia volat odpovedajucu MPI funkciu***
- Obecne sa kolektivne operacie nechovaju ako bariery, tj jeden process moze dokoncit volani funkce drive, nez jiny process vobec dosiahne miesta volania kolektivnej operacie
- Forma virtualnej synchronizacie
- Nepouzivaju *tagy*(vsetci vedia aka operacia sa vykonava)
- Pokial je nutne specifikovat zdrojovy, ci cielovy process, musia tak urobit vsetky participujuce procesy a jejich volba cieloveho ci zdrojoveho procesu musia byt zhodna
- MPI podporuje 2 varianty kolektivnych operacii
    - Posielaju sa rovnake velke data(napr MPI_Scatter) -> nevektorove
    - Posielaju sa rozne velke data(napriklad MPI_Scatterv) ->vektorove
- MPI_Scatter() -> distribuuje mnozinu sprav do zdielaneho prostredia/siete

Bariera

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

- Realizuje bareru

- Volanie funkcie skonci pokial vsetky zucastnene procesy zavolaju MPI_Barrier
- Neni nutne aby volana funkcia bola na “rovnakom mieste v programe”

Kolektivni komunikacni primitive a MPI

Operace	Jméno MPI funkce
OTA vysílání	MPI_Bcast
ATO redukce	MPI_Reduce
ATA vysílání	MPI_Allgather
ATA redukce	MPI_Reduce_scatter
Kompletní redukce	MPI_Allreduce
Gather	MPI_Gather
Scatter	MPI_Scatter
ATA zosobněná komunikace	MPI_Alltoall

Vsesmerove vysielanie – OTA

```
int MPI_Bcast(void *buf, int count,MPI_Datatype
datatype,int source, MPI_Comm comm)
```

- Rozosielane data ulozene v bafru *buf* procesu *source* ostatnym procesum v domene *comm*
- Krome *buf* musia byt parameter funkcie zhodne vo vsetkych participujucich procesoch
- Parameter *buf* na ostatnych procesoch sluzi pre identifikaciu bafru pre prijem dat

Redukce

```
int MPI_Reduce(void *sendbuf, void *recvbuf,int count,
MPI_Datatype datatype,MPI_Op op, int target,MPI_Comm
comm)
```

- Data ze sendbuf zkombinovane operaciou op do *recvbuf* procesu *target*
- Vsetci zucastneni musia poskytnut *recvbuf* I ked vysledok ulozeny iba na procesu target
- Hodnotny *count, datatype, op, target* musia byt zhodne vo vsetkych volajucich procesoch
- Moznost definovat vlastnu operaciu typu *MPI_Op*

Operatory redukce

Operace	Význam	Datové typy
MPI_MAX	Maximum	C integers and floating points
MPI_MIN	Minimum	C integers and floating points
MPI_SUM	Součet	C integers and floating points
MPI_PROD	Součin	C integers and floating points
MPI_LAND	Logické AND	C integers
MPI_BAND	Bitové AND	C integers and byte
MPI_LOR	Logické OR	C integers
MPI_BOR	Bitové OR	C integers and byte
MPI_LXOR	Logické XOR	C integers
MPI_BXOR	Bitové XOR	C integers and byte
MPI_MAXLOC	Maximum a minimální pozice s maximem	Datové dvojice
MPI_MINLOC	Minimum a minimální pozice s minimem	Datové dvojice

MAXLOC -> prva slozka dvojice je maxlock samotny, druhe je ID procesu na ktorom to maximum bolo

MINLOC analogicky

Datove pary a kompletne redukcie

MPI datové páry	C datový typ
MPI_2INT	int, int
MPI_SHORT_INT	short, int
MPI_LONG_INT	long, int
MPI_LONG_DOUBLE_INT	long double, int
MPI_FLOAT_INT	float, int
MPI_DOUBLE_INT	double, int

- Druhym argumentum je int pretoze *rank* je vzdy int

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,int
count, MPI_Datatype datatype,MPI_Op op, MPI_Comm comm)
```

Prefixovy sucet

```
int MPI_Scan(void *sendbuf, void *recvbuf,int count,
MPI_Datatype datatype,MPI_Op op, MPI_Comm comm)
```

- Vykonava prefixovu redukciu

- Process s rankom **I** ma vo vysledku hodnotu vzniknu redukciu procesov s rankom **0 az I** vcetne
- Inak zhodne s redukciou

```
int MPI_Gather(void *sendbuf, int
sendcount,MPI_Datatype senddatatype,void *recvbuf, int
recvcount,MPI_Datatype recvdatatype,int target,
MPI_Comm comm)
```

- Vsetci posielaju stejny typ dat
- Cielovy process obdrzi **p** bafru zoradenych podla ranku odosielatela
- *Recvbuf, recvcount, recvbuf* platne pouze pre process s rankem target
- *Recvcount* je pocet odoslanych dat jednym procesom, nikoliv celkovy pocet prijmacich dat

Allgather

```
int MPI_Allgather(void *sendbuf, int
sendcount,MPI_Datatype senddatatype,void *recvbuf, int
recvcount,MPI_Datatype recvdatatype,MPI_Comm comm)
```

- Bez urcenia cieloveho procesu, vysledok obrdzia vsetci
- *Recvbuf, recvcount, recvdatattype* musia byt platne pre vsetky volajuce procesu

Vektorova varianta Gather

```
int MPI_Gatherv(void *sendbuf, int
sendcount,MPI_Datatype senddatatype,void *recvbuf, int
*recvcounts,int *displs,MPI_Datatype recvdatatype,int
target, MPI_Comm comm)
```

- Odosielatelia mozu odosielat rozne velke data(Rozne hodnoty sendcount)
- Pole *recvcounts* udava, kolko dat moze byt prijate od jednotlivych procesov
- Pole *displ*s udava kde v bafru *recbuf* zacinaju data jednotlviych procesov
    - 0412 -> 0 rank ma 4 bajty, process s rankom 1 8 bajtov, 12 hovori kam skocit

Scatter

```
int MPI_Scatter(void *sendbuf, int
sendcount,MPI_Datatype senddatatype,void *recvbuf, int
recvcount,MPI_Datatype recvdatatype,int source,
MPI_Comm comm)
```

- Posielaju rozne data STEJNEJ velkosti vsetkym procesom

```
int MPI_Scatterv(void *sendbuf, int *sendcounts,int
*displs,MPI_Datatype senddatatype,void *recvbuf, int
recvcounts,MPI_Datatype recvdatatype,int source,
MPI_Comm comm)
```

- Posielaju rozne data ROZNEJ velkosti vsetkym procesom

Alltoall

```
int MPI_Alltoall(void *sendbuf, int
sendcount,MPI_Datatype senddatatype,void *recvbuf, int
recvcount,MPI_Datatype recvdatatype,MPI_Comm comm)
```

- Posiela stejne velke casti bafru sendbuf jednotlivym procesom v domene comm
- Process **i** obdrzi cast velkost sendcount ktora zacina na pozicii sendcount*i
- Kazdy process ma v bafru *recvbuf* na pozici recvcount * I data velkosti recvcount od procesu i

Vektorova varianta Alltoall

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts,int
*sdispls,MPI_Datatype senddatatype,void *recvbuf, int
*recvcounts,int *rdispls,MPI_Datatype
recvdatatype,MPI_Comm comm)
```

- Pole `sdispl` určuje, kde začínají v bafru `sendbuf` data určená jednotlivým procesům.
- Pole `sendcounts` určuje množství dat odesílaných jednotlivým procesům.
- Pole `rdispls` a `recvcounts` udávají stejné informace pro přijatá data.

Skupiny, komunikatory a topologie

```
int MPI_Comm_split(MPI_Comm comm, int color, int
key,MPI_Comm *newcomm)
```

- Kolektivni operace, musi byt volana vsemi
- Parameter *colour* urcuje vyslednu skupinu/domenu
- Parameter *key* urcuje rank vo vyslednej skupine
    - Pri zhode *key* rozhoduje povodny rank

- Stvrty argument je navratova hodnota pre novy argument pre danu domenu(biele/cervine/etc)

Mapovani procesu

Nevyhody "rucneho" mapovania

- Pravidla mapovania urcene v dobe kompilacie programu
- Nemusia odpovedat optimalnemu mapovaniu
- Nevhodne najma v pripadoch nehomogenneho prostredia

Mapovanie cez MPI

- Mapovanie urcene za behu programu
- MPI knihovna ma k dispozicii(aspon ciastocnu) informaciu o sietovom prostredi(napriklad pocet pouzitych procesorov v jednotlivych participujucich uzloch)
- Mapovanie navrhnute s ohladom na minimalizaciu ceny komunikacie

Kartezske topologie

```
int MPI_Cart_create (MPI_Comm comm_old, int ndims, int
*dims,int *periods, int reorder, MPI_Comm *comm_cart)
```

- V tejto funkcie sa specifikuje na kolko skupin/ako velkych skupin sa maju procesy rozdelit a MPI sa posnazi najst take mapovanie aby vyhovelo a minimalizovalo ceny komunikacii atd
- Pokial je v povodnej domene *comm_old* dostatocny pocet procesorov, tak vytvoru novu domenu *comm_cart*s virtualni kartezskou topologii
- Funkciu musia zavolat vsetky procesy z domeny comm_old
- Parametry kartezskej topologie
  - Ndsims – pocet dimenzii
  - Dims[] – pole rozmerov jednotlivych dimenzii
  - Periods[] – pole priznakov cyklickej uzavrenosti dimenzii
- priznak *reorder* znaci ze ranky procesov sa maju v ramci novej domeny vhodne preusporiadat
- nepouzite procesy oznacene rankem *MPI_COMM_NULL*

Koordinaty procesorov v Kartezske topologiech

- komunkacne primitive vyzaduju rank adresata
- preklad z koordinatu(cords[]) do ranku

```
int MPI_Cart_rank (MPI_Comm comm_cart,int *coords, int
*rank)
```

- preklad z rank una koordinaty
- maxdims je velkost vstupneho pola coords[]

```
int MPI_Cart_coord(MPI_Comm comm_cart,int rank, int maxdims,int *coords)
```

Delenie kartezskych topologii

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,MPI_Comm *comm_subcart)
```

- pre delenie kartezskych topologii na topologie s mensou dimenziou
- pole priznakov *keep_dim* urcuje, ci bude odpovedajuca dimenze zachovana v novom deleni

Priklad

- topologie o rozmeroch 2x4x7
- hodnotou keep_dims={true,false,true}
- vzniknu 4 nove domeny o rozmeroch 2x7

Pripadova studie implementace verifikacniho nastroje DiVinE

Divine-cluster

- softwarovy anstroj pre verifikaciu protokolov(LTL MC)
- problem detekcie akceptujuciho cyklu v grafu
- algoritmy paralelne prochazi graf konecneho automatu
- standarni pruzkumova dekompozice

Algoritmus MAP

- detekuje zda existuje akceptujuci vrchol, ktery je svuj vlastni predchudze

Algoritmus OWCTY

- oznacuje vrcholy, ktere nejsou sucastou akceptujuceho cyklu
  - nemaju priamych predchodcov – nelezi na cyklu
  - nema akceptujucich predkv – nelezi na akceptujucom cycle

10 – Analyticky model paralelnych programov

Vyhodnocovanie sekvencnych algoritmov

- casova a priestorova zlozitost
- funkcie a velkost vstupu

vyhodnocovanie paralelnych algoritmov

- paralelny system = algoritmus + platforma
- slozitost
- prinos paralelizmu
- prenositelnost
- atd

Rezie(overhead) paralelnych programov

N-nasobne zrychlenie

- s pouzitim n-nasobnych HW zdrojov, lze ocekavat n-nasobne zychlenie vypoctu
- prakticky nastava zriedka z dovodu rezii paralelneho riesenie
- pokial nastane, je jeho dovodom casto neoptimalne rieseni sekvencneho algoritmu(moze byt aj viac nez n-nasobne zrychlenie pre n-nasobne zdroje)

Duvody

- rezie – interakce, prostoje, zlozitost paralelneho algoritmu
- nerovnomene zvysovanie HW zdrojov
    - zvysenie poctu procesorov nepomoze, pokial aplikacia neni zavisla na cistom vypocetnom vykone

Komunikace

- cena samotnej komunikacie
- priprava dat k odoslaniu, zpracovanie prijatych dat

Prostoje(lelkovani, Idling)

- nerovnomerna zataz na jednotlive vypocetne jadra
- cakanie na zdroje ci data
- nutnost synchronizacie asynchronnych vypoctov

vacsia vypocetna zlozitost paralelneho algoritmu

- sekvencny algoritmus nejde paralelizovat(DFS postorder)
- typicky existuje viacero paralelnych algoritmov, je nutne charakterizovat, cim sa plati za paralelizmus

Metriky vykonnosti

Cas vypoctu – execution time

Otazky

- ako merit vykon/kvalitu paralelneho algoritmu
- podla coho urcit najlepsi/najvhodnejsi algoritmus pre danu platformu

Cas vypoctu – sekvencny algoritmus

- doba, ktora uplynie od spustenia vypoctu do jeho ukoncenia
- T_S

Cas vypoctu – paralelny algoritmus

- Doba, ktora uplynie od spustenia vypoctu do doby, kedy skonci posledny z paralelnych vypoctov
    - Zapocitava sa aj cas napriklad distribuovania dat medzi participujucich procesov ci zbierania vysledkov, ak sa vysledok uklada do suboru niekde na server, rata sa aj to
- Zahrnuje distribuciu vstupnych I zber vystupnych dat
- T_P

Celkova rezia paralelizmu(Total overhead)

Celkova rezia

- Oznacujeme TO
- Veskere element sposobujuce reziu paralelneho riesenia
- Celkovy cas paralelneho vypoctu bez casu potrebneho pre vypocet problem optimalnym sekvencym riesenim
- Paralelne a sekvencne algoritmy mozu byt zcela odlisne
- Sekvencnych algoritmu moze existovat viac
- TS cas vypoctu najlepsieho sekvencneho algoritmu(riesenie)

Funkcia celkovej rezie

- Jaky je cas vypoctu jednotlivych procesov
- Doba po skonceni vypoctu jedneho procesu do skoncenia celeho paralelneho vypoctu sa povazuje za reziu (idling)
- TO = pTp – TS
    - p je pocet vlakien, Tp paralelny cas, Ts optimalnysekvencny cas

zrychlenie (speed-up)

Zakladny prinos paralelizacie

- problem idu vzdy riesit sekvencnym algoritmom
- paralelizaciou lze dosiahnut iba zrychlenie vypoctu
- ostnatne vyhody su diskutabilne a tazko meritelne

Zakladna miera ucinnosti samotnej paralelizacie

- pomer casu potrebnych k vyrieseniu ulohy na jednej procesnej jednotke a p procesnych jednotkach
- uvazuje sa vzdy cas najlepsieho sekvencneho algoritmu

- v praxis sa casto(a nespravne) pouziva cas potrebny k vyrieseniu ulohy paralelnym algoritmom spustenym na jednej procesnej jednotke
- S = TS /TP

Teoreticka hranica zrychlenia

- S pouzitim **p** procesnych jednotkej je maximalne zrychlenie **p**

Super-linearne zrychlenie

- Jav, kedy zrychlenie je vacsie ako **p**

Pozorovanie

- Zrychlenie lze merit I asymptoticky

Priklad – scitani n cisel s pouzitim n procecesnich jednotiek

- Sekvenčně je potřeba Θ(n) operací, časΘ(n)
- Paralelně je potřeba Θ(n) operací, ale v časeΘ(log n)
- Zrychlení S= Θ(n/log n)

Super-linearne zrychlenie

Falosne super-linearne zrychlenie

- Uvazme datovu distribuciu na 2 procesne jednotky a operaciu kvadraticku vo velkosti dat
- Zrychlenie S = n^2 /((n/2)^2) pri pouziti 2 procesnych jednotiek
- Problem – neuvazovany optimalny sekvencny algoritmus

Skutocne super-linearne zrychlenie

- Vacsia agregrovana velkost cache pameti
- Pri znizeni mnzostva dat na jeden procesny uzol sa ucinost cache pamete zvysi
- Vypocet je na 1/p datach viac ako p-krat rychlejsi, pretoze process nemusi stale siahat do pameti

Super-linearne zrychlenie zavislosti na instancii problem

- Priezkumova dekompozicia ulohy pri hladani riesenia
    - Superlinearne zrychlenie je priezkumovej dekompozicii vlastne
- Paralelna verzia moze vykonat mensie mnozstvo prace
- V konkretnej instancii lze paralelne prehladavanie simulovat I sekvencnym algoritmom, obecne ale nelze

Amdahlov zakon

Otazka

- Ake je najvacsie mozne zrychlenie system, pokial sa paralelizaciou urychli iba cast vypoctu

Amdahluv zakon

- $S_{max}=1/((1-P)+(P/Sp))$
- P – podil systemu, ktory je urychleny paralelizaciou
- Sp – zrychlenie dosiahnute paralelizaciu nad danym podielom

Příklad

- Paralelizací lze urychlit 4-násobně 30% kódu, tj. $P = 0.3$ a $S_P = 4$.
- Maximální celkové zrychlení S_{max} je

$$S_{max} = \frac{1}{(1-0.3)+\frac{0.3}{4}} = \frac{1}{0.7+0.075} = \frac{1}{0.775} = 1.29$$

Efektivita

Fakta

- Pouze idealni paralelni system s **p** procesnymi jednotkami moze dosahovat zrychlenie **p**
- Cast vypoctu vykonavana na jednom procesore je spotrebovana reziou. Procesor nevenuje 100% vykou riesenia problem

Efektivita

- Lze definovat ako podiel zrychlenia S voci poctu jednotiek **p**
- E = S/p = (TS/TP)/p = TS/(p*Tp)
- Zrychlenie je v praxis <p, efektivita v rozmezi (0,1>
- “podiel casu, po ktory jednotka vykonava uzitocny kod”
- Vo vedeckej komunity najviac pouzivane pre porovnavanie kvality paralelnych algoritmov

Příklad 1

- Jaká je efektivita sčítání n čísel na n procesorech?
- $S = \Theta(\frac{n}{\log n})$
- $E = \Theta(\frac{S}{n}) = \Theta(\frac{1}{\log n})$

Příklad 2

- Na kolik se zkrátí 100 vteřinový výpočet při použití 12 procesorů při 60% efektivitě?
- $E = \frac{T_s}{pT_p}$
- $0.6 = \frac{100}{12x} \implies x = \frac{100}{0.6*12} \implies x = 13.88$

Cena

Cena riesenia problem na danom paralelnom system

- Sucin poctu procesnych jednotiek a doby paralelneho vypoctu:
  - C = p x Tp
- Oznacovana take “mnozstvo prace”, ktore paralelny system vykona, nebo taktiez “pTp product”
- Alternativne lze pouzit pre vypocet ucinnosti(E= Ts/C)

Cenovo optimalny paralelny system

- Cena sekvencneho vypoctu odpoveda najlepsiemu Ts
- Paralelny system je cenovo optimalny pokial cena riesenia rastie asymptoticky stejne rychle ako cena sekvencneho vypoctu
- Cenovo optimalny system musi mat efektivitu rovnu O(1)

Priklad

- Scitanie n cisiel na n procesoroch
- C = O(n*logn), neni cenovo optimalny

- Uvažme n procesorový systém, který třídí (řadí) seznam n čísel v čase $(log\ n)^2$
- Nejlepší sekvenční algoritmus má $T_S = nlog\ n$
- $S = \frac{n}{log\ n}$, $E = \frac{1}{log\ n}$
- $C = n(log\ n)^2$
- Není cenově optimální, ale pouze o faktor $log\ n$
- Uvažme, že v praxi $p << n$ (p je mnohem menší než n)
- $T_P = n(log\ n)^2/p$ a tedy $S = \frac{p}{log\ n}$
- Konkrétně: $n = 2^{10}$, $log\ n = 10$ a $p = 32 \Rightarrow S = 3.2$
- Konkrétně: $n = 2^{20}$, $log\ n = 20$ a $p = 32 \Rightarrow S = 1.6$
- Závěr: cenová optimalita je nutná

Vliv granularity na cenovu optimalitu

Tvrzeni

- Volbou granularity lze ovlivnit cenu paralelneho riesenia

Pozorovanie

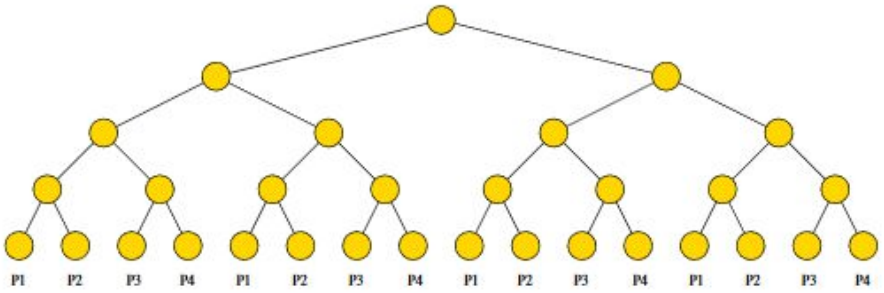
- V praxis p<<n presto navrhujeme algoritmy tak, aby granularita bola az na urovni jednotlivych polozek vstupu, tj predpokladame ze p==n

Deskalovanie(scale-down)

- Umele znizujeme granularitu(vytvarime hrubsie ulohy)
- Znizujeme overhead spojeny s komunikaciou
- Moze ovlivnit cenu a cenovu optimalitu

Úkol

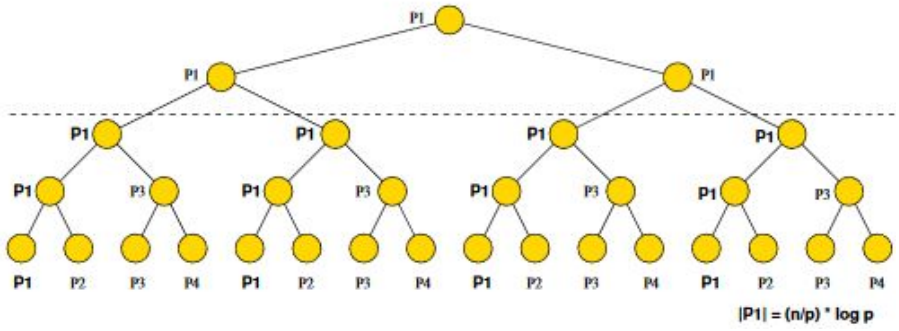
- Sčítání n čísel na p procesorech ($n = 16$, $p = 4$)
- Algoritmus „stromového“ sčítání.
- Mapování ($i \ mod \ p$), tj. (n/p) čísel na 1 procesor.



Vplyv granularity na cenovu optimalitu

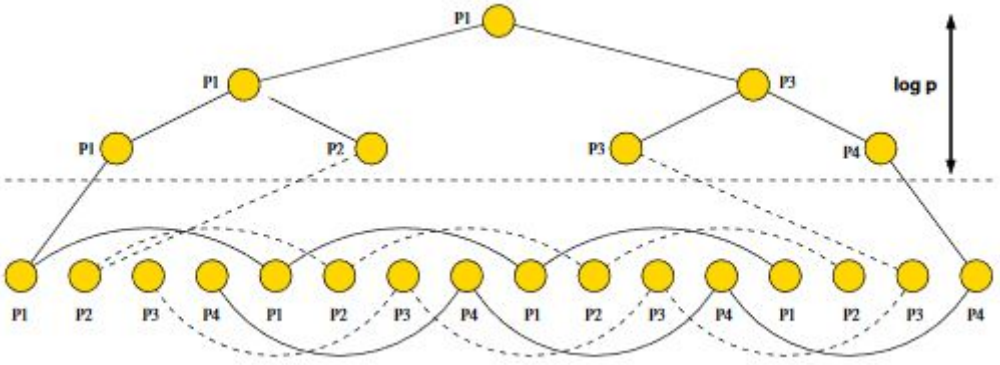
Analýza ceny

- První část výpočtu proběhne v čase $\Theta((n/p)log\ p)$.
- Zbylá část výpočtu probíhá lokálně v paměti jednoho procesoru v čase $\Theta(n/p)$.
- Celkový T_P je $\Theta((n/p)log\ p + n/p) = \Theta((n/p)log\ p)$
- Cena $C = p * T_p = \Theta(nlog\ p)$, tj. není cenově optimální



Deškálovaný algoritmus a analýza jeho ceny

- Každá jednotka nejprve sečte data lokálně v čase $\Theta(n/p)$
- Problém redukován na sčítání p čísel na p procesorech
- Celkový T_P je $\Theta(n/p + log\ p)$
- Cena $C = \Theta(n + plog\ p)$,
- Cenově optimální, pokud $n > p\ log\ p$ (cena $C = \Theta(n)$)



Skalovatelnost paralelnych programov

Skalovatelnost

- Zachovanie vykonu a efektivity pri zvysovani poctu procesnych jednotiek a zvacsovania objemu vstupnych dat

Pozorovanie

- Programy su testovane na malych vstupnych detach na systemoch s malym poctom procesnych jednotiek

- Algoritmus ktory vykazuje najlepsi vykon na testovanych detach, sa moze ukazat byt tim najhorsim algoritmom pri pouziti na skutocnych datach
- Odhad vykonu aplikacie nad realnymi vstupnymi datmi a vacsim poctom procesnych jednotiek je komplikovany

Zaver

- Je treba uvazit analyticke techniky pre vyhodnotovanie vykonu a skalovania

Charakteristiky skalovatelnosti - efektivita

Efektivita paralelních programů:

$$E = \frac{S}{p} = \frac{T_S}{pT_P} = \frac{1}{1 + \frac{T_O}{T_S}}$$

Celkova rezia (TO)

- Pritomnost sekvencnej casti kodu je nevyhnutelna. Pre jej vykonanie je treba cast serial. Po tuto dobu su ostatne procesne jednotky nevyuzite
- TO > (p-1)t*serial*
- TO rastie minimalne linearne vzhladom k p, castokrat aj asymptoticky viac

Uloha konstantnej velkosti (TS fixni)

- So zvysujucim sa **p**, efektivita nevyhnutelne klesa
- ***Nevyhnutely udel vsetkych paralelnych programov***

Problém

- Úkol: Sečíst n čísel s využitím p procesních jednotek.
- Uvažme cenově optimální verzi algoritmu.
- $T_p = (\frac{n}{p} + log\ p)$.
- Lokální operace stojí 1, komunikace 2 jednotky času.

Charakteristiky řešení

- $T_P = \frac{n}{p} + 2log\ p$
- $S = \frac{n}{\frac{n}{p}+2log\ p}$
- $E = \frac{1}{1+\frac{2plog\ p}{n}}$

Charakteristiky skalovatelnosti – skalovatelny system

Pri zachovani Ts

- Rastuci pocet procesnych jednotiek znizuje efektivitu

Pri zachovani poctu procesnych jednotiek

- Rastuci Ts(velkost vstupnych dat) ma tendency zvysovat efektivitu

Skalovatelne systemy

- Efektivitu lze zachovat pri subeznom zvysovani Ts a **p**
- Zajimame sa o pomer, v ktorom sa TS a **p** musia zvysovat, aby sa zachovala efektivita
- Cim mensi je pomer TS/p, tym lepsia skalovatelnost

Izoefektivita ako metrika skalovania

Vztahy

- $T_P = \frac{T_S + T_O(W,p)}{p}$ W – velikost vstupních dat
- $S = \frac{T_S}{T_P} = \frac{pT_S}{T_S + T_O(W,p)}$
- $E = \frac{S}{p} = \frac{T_S}{T_S + T_O(W,p)} = \frac{1}{1 + T_O(W,p)/T_S}$

Konstantní efektivita

- Efektivita je konstantní, pouze pokud $T_O(W,p)/T_S$ je konstantní.
- Úpravou vztahu pro efektivitu: $T_S = \frac{E}{1-E} T_O(W,p)$.
- Při zachování míry efektivity lze $E/(1-E)$ označit jako konstantu K.

Funkce izoefektivity (stejné efektivity)

$$T_S = KT_O(W,p)$$

Pozorovani

- Pri konstantnej efektivite lze TS vyjadrit jako funkci **p**
- Vyjadruje vztah, ako sa musi zvysit Ts pri zvyseni p
- Nizsi funkce rika, ze system je snaze skalovatelny
- Isoefektivita nelze merit u system ktore neskaluju

Příklad

- Součet n čísel na p procesorech
- $E = \frac{1}{1+(2p \log p)/n} = \frac{1}{1+T_O(W,p)/T_S}$
- $T_O = 2p \log p$
- Funkce izoefektivity: $T_S = K2p \log p$
- Funkce izoefektivity: $\Theta(p \log p)$

- Při zvýšení procesních jednotek z p na p' se pro zachování efektivity musí zvětšit velikost problému o faktor $(p' \log p')/(p \log p)$.

Cenova optimalita a izoefektivita

Optimalni cen

- Pre cenove optimalne systemy pozadujeme pTP = O(TS)
- Po dosadeni zo zakladnych izo vztahov dostavame
  - TS + TO(W,p) = O(TS)

Cenova optimalita moze byt zachovana pouze ak:

- Rezie je najvyse linearni voci zlozitosti sekvencneho algoritmu, tj funkce TO(W,p) $\in$O(Ts)
- Slozitost sekvencneho algoritmu je minimalne tak velka ako rezie, tj funkce Ts$\in\Omega$(To(mW,p))

Spodni odhad na izoefektivitu

Izoefektivita

- Cim nizsi/mensi funkce, tim lepsi skalovatelnost
- Snaha o minimalnu hodnotu izofunkce

Sublinearni izoefektivita

- Pre problem tvoren n jednotkami prace, je otpimalna cena dosiahnute pre maximalne N procesnych jednotiek
- Pridavam dalsich jednotiek vyusti v idling niektorych jednotiek a znizovanie efektivity
- Aby sa efektivita nesnizovala, tak musi mnozstvo prace rast minimalne linearne vzhladom k p
- Funkce izoefektivity nemuse byt sublinearni

Minimalni doba vypoctu

Otázka

- Jaká je minimální možná doba výpočtu (T_P^{min}) při dostupnost dostatečného počtu procesních jednotek?

Pozorování

- Při rostoucím p se T_P asymptoticky blíží k T_P^{min}.
- Po dosažení T_P^{min} se T_P zvětšuje spolu s p.

Jak zjistit T_P^{min}?

- Nechť p_0 je kořenem diferenciální rovnice $\frac{dT_P}{dp}$.
- Dosazení p_0 do vztahu pro T_P dává hodnotu T_P^{min}.

Příklad – součet n čísel p procesními jednotkami

- $T_P = \frac{n}{p} + 2log\ p, \quad \frac{dT_P}{dp} = -\frac{n}{p^2} + \frac{2}{p}$
- $p_0 = \frac{n}{2}, \quad T_P^{min} = 2log\ n = \Theta(log\ n)$

Asymptoticka analyza paralelnych programov

Algoritmus	A1	A2	A3	A4
p	n^2	$log\ n$	n	$\sqrt{n}$
T_P	1	n	$\sqrt{n}$	$\sqrt{n}log\ n$
S	$n\ log\ n$	$log\ n$	$\sqrt{n}log\ n$	$\sqrt{n}$
E	$\frac{log\ n}{n}$	1	$\frac{log\ n}{\sqrt{n}}$	1
pT_P	n^2	$n\ log\ n$	$n^{1.5}$	$n\ log\ n$

Otázky

- Jaký algoritmus je nejlepší?
- Použily jsme vhodné odhady složitosti?
- Je dostupná odpovídající paralelní architektura?