**Subežnosť** : Existencia dvoch a viacerých procesov v jednom okamžiku

Vzpočetný paralelismus,Výkonnosť,Uskutočnitelnosť,Bezpečnosť,Cena

Abstraktný model výpočtového systému:

Procesor – Dátová cesta – Dátové úložisko

* Všetky miesta môžu byť uzkym miestom voči výkonnosti aplikácie ako celku
* Paralelizmus je prirodzený spôsob prekonania týchto uzkých ciest

Obmädzenia procesorov:

* Moore : 2x tranzistorov kazdých 18 mesiacov
* Maximalne zmenšenie na 5nm

Riešenie:

* Multi-core, Many-core procesory
* Hybridné riešenia

Následok:

* Sekvenčné algoritmy už nemôžu ťažiť z výkonu procesoru
* Paralelizacia je nevyhnutný smer vývoja

Komunikácia a paralelismus:

* Väčšia priepustnosť komunikačných liniek => snižovanie latencie
* Robustnosť a spoľahlivosť kom. Liniek

Príklad:

* Šírka zbernice : 32/64/128 bitov
* Domáci dual-band WIFI router

Pamäť:

* Výkon procesoru prevyšuje výkon ostatných komponentov
* Cesta CPU – RAM- HDD je zdĺhavá
* Doba získaní informace rastie so vzdialenosťou miesta uloženia od miesta spracovania

Úrovne uloženia Informácii:

* Registre CPU
* L1/L2/L3 cache
* Operačná pamäť
* Cache I/O
* Magnetické/ Optické mechaniky

Cache:

* Kopia časti dat v rýchle dostupnom mieste
* Môže – nemusí byť kontrolovatelná programátorom
* L1/L2 nekontrolovatelná
* I/O efficient algoritmy: obchádzanie virtualizácie pamati, namiesto toho realizácia vlastného spôsobu využitia RAM ako cache I/O zariadení

Mnoho pamäťových modulov:

* Väčšie množstvo uložitelných/zapamätatelných info.
* Väčšie množstvo liniek do pamäti (Priepustnosť)
* Väčšia réžia na udržani konzistencie
* Príklad: Diskové polia, Peer-to-Peer siete

# Flynnova klasifikácia

* SISD Single Instruction Single Data, jedna inštrukcia nad jedným datovým prúdom (sekvenčný prístup)
* SIMD Single Instruction Multiple Data, jedna inštrukcia nad viacerými dátovými prúdmi

(Vektorové inštr. CPU, architektúra GPU)

* MIMD Multiple Instruction Multiple Data, nezávislý súbeh dvoch a viac SISD a SIMD prístupov

(Viac jadrové procesory)

* MISD Multiple Instruction Single Data, praktickz sa nevyskytuje

# Distribuovaný systém

* Špecifikovaný po častiach => Procesoch
* Chovanie systému vzniká interakciou súbežných procesov
* Emergentné javy
* Synchronizácia: Obmädzenie na prekladanie a súbeh akcií jednotlivých procesov distrib. Systemu
* Komunikácia: Prenos informácie z jedného procesu na druhý
* Problémy Distribuovaného systému:
* Nekonzistetná vízia konzistentného sveta
* Vzájomná interferencia
* Race-condition, nedeterministrické chovanie
* Uviaznutie – Deadlock,Livelock
* Starwing – stárnutie
* Buffer-overflow, problém producent-konzument
* Active waiting – zbytočné strata výkonu
* Dôvody Náročnosti vývoja paralel. aplikácií:
* Nutnosť špecifikácie súbežných úloh a ich koordinácia
* Paralelné algoritmy
* Nedostačujúce IDEs
* Nedeterminizmus pri simulácii aplikácie
* Rýchly vývoj a zostarnenie použitých technológii
* Výkon aplikácie náchylný na zmenu v konfigurácii systému

# HPC

* High Performance computing

# Programovanie v prostredí so sdieľanou pamäťou

* Systémy s viac procesormi, Systémy s viac-jadrovými procesormi,Systémy so zabudovanými SMT
* Kombinácia
* Riziká: optimalizácie na úrovni CPU zachovávajú sémantiku sekv. Programov

## Simultánny multithreading SMT

* CPU využíva prázdné cykly zpôsobené latenciou RAM k vykonávaniu iného vlákna
* Vyžaduje duplicitné registre
* Vlákna zdieľaju cache

## Viac-jadrové CPU (multicores)

* **Výhody:** efektívnejšia cache koherencia na najnižšej úrovni,nižšie náklady pre koncového užívatela
* **Nevýhody:** väčšie teplo,nižší takt jedného jadra,auto podtaktovanie/pretaktovanie,zdieľajú datovú cestu do pamäti

# Paralelismus Ideálny model a Realita

* Idealizovaný model:
* Na tejto úrovni sa rieši návrh paralel. Algorimu
* Jednotlivé výpočet. Jadrá pracujú nezávisle
* Prístupy k datám v pamäti sú bezčasové a nezávislé
* Komunikácia úloh prebieha atomicky cez zdieľané dátové štruktúry
* Realita:
* Na tejto úrovni rieši programátor technickú realizáciu
* Prístup do pamäti cez zbernicu pre CPU velmi pomalý
* Problém koherencie dát
* Registre procesoru a cache pamäti – rýchle kópie malého množstva dát

# Procesy a vlákna

Procesy:

* Skrývajú pred ostatnými procesmi svoje výpočetné prostriedky
* Pre riešenie paralelnej úlohy je treba medzi-procesová komunikácia **IPC**

Zdieľané pamätové segmenty, sokety, named/non-naimed pipes

Vlákna:

* Existujú v kontexte jedného procesoru
* V rámci rodičovského procesu zdieľajú výpočetné prostriedky
* Komunikácia prebieha cez zdieľané dátové štruktúry
* Účelom interakcie je skôr sznchronizácia než transport dát
* Realizuje výpočet, sekvenciu inštrukcii
* Každý proces je tvorený aspoň jedným vláknom
* Hlavné vlákno vytvára ďalšie

## Procesy:

* Identifikátory procesu a vlastníka, premenné prostredia, pracovný adresár, kód,
* Registre, Zásobník, Halda
* Odkaz na otvorené súbory a zdieľané knihovne
* Reakcie na signály
* Kanály IPC

## Vlákna:

* Majú privátne: Zásobník,Registre
* Fronta signálov
* Vytvorenie vlákna je lacnejšie, zmena dát vo vlákne je viditelná v rámci celého procesu
* Komunikácia spočíva v predávaní referencie na data a nie predávaním obsahu
* Predávanie referencie sa deje v rámci jedneho procesu OS nemusí riešiť skrývanie dát

# Efektívne využitie cache

* Menšia ale rýchlejšia pamať na pomalej dátovej ceste
* Okolie čítanej informácie sa uloži do cache
* Pri nasledujúcom čítaní sa už informácie čítajú z cache
* Cache line – atomický pamätový blok uložený v cache
* Hit ratio – číslo vyjadrujúce úspešnosť obslúženia požiadavkov na data uložené v cache
* Vyliate cache – procedura aktualiyacie dat v pamati hodnotami uloženými v cache
* Efektívne využívanie cache:
* Časová lokalita: prístupy v malom časovom interval
* Priestorová lokalita: prístup k datám uložených adresne blízko seba
* Zarovnaná alokácia pamäte

# Koherencia

* Súlad dát uložených v pamäti počítača (hlavne v cache)
* Zaisťuje že existuje len jedna platná hodnota asociovaná s daným pamäťovým miestom
* Zápisy CPU do pamäti sú zdržované a odkladané kvôli rýchlosti

# False sharing

* Paralelná cache koherentný systém s viac procesormi
* Program s niekolkonásobne viacero vláknami

## Nestála premenná

* Pred a po každom použití v cache zapísať do pamäte
* Nestála premenná: volatile T prem, T volatile prem
* Ukazateľ na nestálu premennú: volatile T \* prem
* Nestály ukazateľ na nestálu premenú: volatile T \* volatile prem
* Použitie:
* Premenná použitá medzi súbežnými vláknami / procesmi
* Zastupuje I/O
* Modifikovaná procedurou obsluhujúcu prerušenie
* Prehnaná dôvera v Volatile:
* Pričítanie nieje realizované jednou inštrukciou -> preloženie inštruckií
* Zápisy hodnôt do pamäte nie su realizované v dobe vykonania inštrukcie
* Záleži na pamäťovom modeli CPU
* Poradie zápisu v CPU =/= zápisu v pamäti

# Pamäťová bariera

* HW primitívum na synchronizáciu stavu pamäti a CPU v danom mieste program
* Inštrukcia **mfence** na súčasných počítačoch
* Vykoná všetky load/store inštrukcie pred príkazom mfence a tento efekt bude viditelný globálne
* Nerieši problém atomických inštrukcíí TEST-AND-SET,COMPARE-AND-SWAP
* HW podpora: x86 **lock** nasledujúca inštrukcia prebehne atomicky a výsledok bude viditelný globálne,**XCHG** prehodí obsah registrov a pamätového miesta

# Race condition

* Nedokonalosť paralelného programu, prejavuje sa nedeterministickým chovaním

# Atomicita operácií

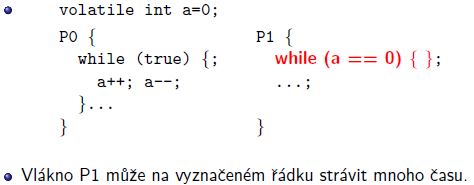
* Jednoduchý príkaz vo vyššom progr. Jazyku nemusí nutne zodpovedať jednej inštrukcii CPU
* Jeden príkaz jazyka môže byt prerušený alebo preložený iným príkazom iného vlákna

# Deadlock

* Ak majú vlákna požiadavky na unikátne zdieľané zdroje môže dojsť k uviaznutiu, nemožnosti pokračovania vo vypočte

# Starving,Livelock

* Jav, kedy je aspoň jedno vlákno neschopné vzhľadom k paralelnému súbehu s iným vláknom pokročiť vo výpočte za danú hranicu



# Thread-safe procedúra a re-entrantná procedúra

* Procedúra ktorej vykonávanie je bezpečné vykonávať súbežne s niekoľkými vláknami bez dodatočnej synchronizácie
* rand() -> rand\_r()
* **Re/entrantná procedúra:** vykonávanie môže byt v rámci jedného vlákna prerušené, kód kompletne vykonaný od začiatku po koniec v rámci rovnakej úlohy potom obnovené prerušenie vykonávania kódu

# Nebezpečné akcie vzhľadom k parallel. Programovaniu

* Nekontrolovaný prístup k glob. Premennej
* Uchovávanie stavu procedúry do glob. Premennej
* Alokácia/Dealokácia zdrojov globálneho rozsahu
* Nepriamy prístup k datám cez odkazy alebo ukazovatele
* **Bezpečná stratégia:**
* Prístup iba k lokálnym premenným (zásobník)
* Kód je závislý iba na argumentoch funkcie
* Všetky volané procedúry sú re-entrantné

## Petersonov algoritmus:

* Vzájomné vylúčenie
* Nespôsobuje stárnutie ani uviaznutie

## Uspávanie:

* Procesy po neúspechu prístupu k zdieľanej premennej sa uspia
* Sú prebudené po časovom limite alebo zobudené iným vláknom

## Spinlock:

* Vlákna opakovane skúšajú prístup do KS
* Efektívnejsie pre krátke čakacie doby než prepínanie kontextu Vlákien/procesov

## Prístup k zdieľaným glob premenným

* Všetký neatomické modifikácie musia byť serializované tj. Uskutočnené po získaní odpovedajúceho zámku na danú operáciu
* Získanie zámkz vynucuje vyliatie cache pameti
* Býva to úzkym miestom výkonu aplikácie

## Thread-private data

* Vlákna majú svoje lok. Premenné
* Data môžu byť uložené v glob. Štruktúre a pristupované iba unikátnym identifikátorom daného vlákna, môže dojsť k **False Sharing**

# POSIX THREAD API

* Funkcionalita:
* Vytváranie, oddeľovanie a spojovanie vlákien
* Nastavenie a zistenie stavu vlákna
* Mutex – vzájomné vylúčenie, vytváranie, ničenie,odomykanie/zamykanie mutexov
* **Condition variable:** Komunikácia/sync vlákien, funkcie na vytváranie,ničenie a signalizovanie pri specifickej hodnote podmienkovej premennej
* **60 funckcií <pthread.h> ,**  preklad **–pthread**
* pthread\_ , pthread\_attr\_
* pthread\_mutex\_ , pthread\_mutexattr\_
* pthread\_condr\_ , pthread\_condattr\_
* pthread\_key\_

# Správa vlákien

* **int pthread\_create**(**pthread\_t\*** thread\_handle, **const pthread\_attr\_t\*** attributes,**void\* (\*thread\_function)**(void\*),**void\*** arg); vytvorí vlákno, ihneď pripravené
* úspešné vytvorenie vlákna je 0
* **void pthread\_exit(void\* value)**,prostriedky sa uvoľňujú až po ukončení rodičovského procesu
* **pthread\_cancel(pthread\_t \* thread\_handle)**
* žiadosť oukončenie vlákna iným vláknom
* Upratanie dát po ukončení vlákna
* Fcia skončí po odoslaní požiadavku, návr hodnota 0 znamená že vlákno existuje
* **pthread\_join(pthread\_t** thread\_handle,**void\*\*** ptr\_value**);**
* čaká na dokončenie vlákna thread\_handle
* ptr\_value ukazateľ na pointer, použije sa pri volaní pthread\_exit(ptr\_value);
* **Detached threads (**nespojiteľné vlákna**):**
* Nemôžu byť spojené volaním pthread\_join
* Šetrí systémové prostriedky
* Doporučené typ vlákna explicitne nastaviť
* **int pthread\_detach(pthread\_t\*** thread\_handle**)**
* **int pthread\_attr\_setdetachstate(pthread\_attr\_t\*** attr, **int** detachstate**);**
* **int pthread\_attr\_getdetachstate(pthread\_attr\_t\*** attr**,int\*** detachstate**);**
* **Vlastnosti vlákien:**
* Veľkosť zásobníka
* int pthread\_attr\_getstacksize (pthread attr t \*attribute, size t \*stacksize)
* int pthread\_attr\_setstacksize (pthread attr t \*attribute, size t stacksize)
* int pthread\_attr\_getstackaddr (pthread attr t \*attribute, void \*\*stackadr)
* int pthread\_attr\_setstackaddr (pthread attr t \*attribute, void \*stackadr)

# Mutex

* **int** **pthread\_mutex\_init**(pthread\_mutex\_t \*mutex lock,pthread\_mutex\_attr\_t \*attribute);
* parametr attribute specifikuje vlastnosti zámku
* NULL znamená počiatočné nastavenia
* **int pthread\_mutex\_lock (pthread\_mutex t \*** mutex\_lock**)**
* zamyká mutex\_lock, volanie je blokujúce dokiaľ sa nepodarí zamknúť, zaberá čas
* Vlákno ktorému sa to podarí je v KS
* Velké KS znižujú výkon Aplikácie
* **int pthread\_mutex\_unlock (pthread\_mutex\_t \***mutex\_lock**);**
* odomyká mutex\_lock
* **int pthread\_mutex\_trylock (pthread\_mutex\_t \*** mutex\_lock**);**
* skúsi zamknúť mutex, úspech == 0, neúspech EBUSY
* má zmysel atívne čakať v cycle volaním **trylock**

## Vlastnosti:

* **Normálny mutex:**
* Iba jedno vlákno môže jedenkrát zamknúť mutex
* Uviaznutie = ak sa to isté vlákno pokúsi zamknúť ten istý mutex
* **Rekurzívny mutex:**
* Dovoľuje jednému vláknu zamknúť ten istý mutex opakovane
* Mutex obsauje čítač zamknutí, nenulový čítač == zamknutý mutex
* Odomknutie = volane unlock tolkokrát kolko bolo volané lock
* **Normálny mutex s kontrolou chyby:**
* Normálny mutex ale pri pokuse o dalšie zamknutie ohlási chybu
* Pomalší ako normálny, zvyčajne použitý len počas vývoja
* **int pthread\_mutexattr\_settype\_np (ptrhead\_mutexattr\_t \***attribute**,int** type**)**
* PTHREAD\_MUTEX\_NORMAL\_NP, PTHREAD\_MUTEX\_RECURSIVE\_NP, PTHREAD\_MUTEX\_ERRORCHECK\_NP

# Condition variables

* Vyžaduje použitie mutexu
* Po získaní mutexu sa ho vlákno môže dočasne vzdať a uspať sa
* Prebudenie je signalizované explicitne iným vláknom
* **int pthread\_cond\_init (ptrhead cond t \***cond**,pthread\_cond\_condattr t \***attr**)**
* Inicializuje podmienkovú premennú, attr == NULL => počiatočné nastavenia
* **Int pthread\_cond\_destroy(pthread\_cond\_t\*** cond**);**
* zničí nepoužívanú podmienkovú premennú a súvisejúce dátové štruktúry
* **int pthread\_cond\_wait (ptrhead\_cond\_t \***cond**,pthread\_mutex\_t \***mutex\_lock**)**
* odomkne mutex\_lock, zablokuje vlákno v spojení s podmienkovou premennou cond
* po návrate vlákno opäť vlastní mutex\_lock
* pred použitím ale musí vlákno mať mutex\_lock zamknuté
* **int pthread\_cond\_signal (ptrhead\_cond\_t \***cond**)**
* signalizuje prebudenie vlákna spiaceho nad danou podmienkovou premennou **cond**
* **int pthread\_cond\_broadcast (ptrhead\_cond t \***cond**)**
* Signalizuje prebudenie všetkých vlákien nad premennou cond
* **int pthread\_cond\_timedwait(ptrhead\_cond\_t \***cond**,pthread\_mutex t \***mutex\_lock**,**

**const struct timespec \***abstime**)**

* Vlákno buď zobudené signálom alebo po uplynutí časového interval
* Po uplynutí času vracia chybu ETIMEDOUT a neimplikuje znovu získanie mutex\_lock

# TSD – Thread specific data

* Globálne premenné, ktoré môžu mať pre každé vlákno inú hodnotu
* Napr. Pole indexované ID číslom vlákna
* **POSIX riešenie:**
* Identifikátor (kľúč) a asociovaná hodnota
* Identifikátor je globálny, asociovaná hodnota je lokálna premenná
* **pthread\_key**
* Asociovaná hodnota – univerzálny ukazateľ tj. **Void\***
* **int pthread\_key\_create (ptrhead\_key\_t \***key**,void (\***destructor**)(void\*))**
* Vytvorí nový kľúč (glob.premenná)
* Destructor – funkcia volaná nad asociovanou premennou daného vlákna v okamžiku ukončenia vlákna, ak nieje nulový, **Nepovinný**
* **int pthread\_key\_delete(ptrhead\_key\_t** key**)**
* nevolá destructor, programátor musí dealokovať objekty pred zničením kľúča
* **void \* pthread\_getspecific (ptrhead\_key\_t** key**)**
* **int pthread\_setspecific (ptrhead\_key\_t** key**,const void \***value**)**
* **ptrhead\_t pthread\_self ()**
* vracia unikátny systémový identifikátor vlákna
* **int pthread\_equal (pthread\_t** thread1**,pthread\_t** thread2**)**
* vráti nenula pri identite vlákien
* **pthread\_once\_t once\_control = PTHREAD\_ONCE\_INIT;**

**int pthread\_once(pthread\_once\_t \***once\_control**,void (\***init\_routine**)(void));**

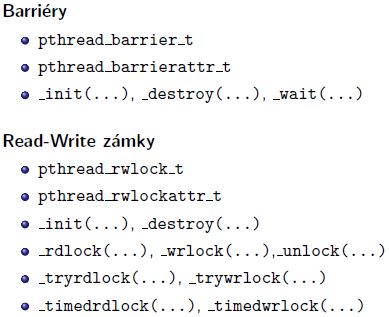
* pri prvom volaní z akéhokoľvek vlákna spôsobí volanie funkcie init\_routine
* ďalšie volania nespôsobia nič

# WRRM Mapa Čitatelia & Písatelia

* Vlákno-čítar môže vtúpiť do KS ak v nej nie je alebo na nu nečaká pisár
* Vlákno-čítar môže vstúpiť do KS ak su v nej iní čítari
* Prístupy vlákien písarov sú serializované a urpednostnené pred čítarmi
* **Riešenie:**
* **rwlock\_rlock(rwlock\_t \*** lock**) -** vstup vlákna čítateľa
* **rwlock\_wlock(rwlock\_t\*** lock**) –** vstup pisateľa
* **rwlock\_unlock(rwlock\_t\*** lock**) –** opustenie ľubovoľného vlákna

# Bariéry

* Synchronizačné primitívum
* Vláknu je dovolené pokračovať po bariére až keď ostatné vlákna dosiahli bariéru
* Implementácia cez mutexy – vyžaduje aktívne čakanie
* Lepšie riešenie je implementácia cez podmienkovú premennú a počítadlo
* Každé vlákno ktoré dosiahlo bariéru zvýši počítadlo, pokial nie je dosiahnutý počet vlákien = podmienené čakanie



# Semafor

* Čítače používané ku kontrole prístupu k zdieľaným premenným
* POSIX semafory – v rámci procesu
* System V semafory – medzi procesmi
* Dá sa použiť aj na synchronizáciu vlákien
* Celočíselý nezáporný čítač – hodnota indikuje obsadenost zdoja
* Nula – zdroj nie je k dispozícií
* Nenula – zdroj nie je využívaný
* **sem\_init()** inicializuje čítač zadanou východziou hodnotou
* **sem\_wait()** zníži čítač pokiaľ môže, inač blokuje
* **sem\_post()** zvýši čítač o 1, zobudí čakajúce vlákno

## Semafory vs. Mutexy

* Mutex smie odomknúť iba to vlákno ktoré ho zamklo
* Semafor môže byť srpavovaný rôznymi vláknami

## Monitor

* Synchronizačné primitivum vyššieho progr. Jazyka
* Označenie kódu ktoré môže byť vykonávané najviac jedným vláknom

## Semafory, mutexy, monitory

* So semaformi a mutexmi je explicitne manipulované programátorom
* Vzájomné vylúčenie realizované monitorom je implicitné tj. Explicitné zamykanie doplní prekladač

## Win32 Vlákna

* Jeden typ HANDLE
* Jedna funkcia pre jednu činnosť
* Udalosti namiesto condition variables

# Implementácia Lock-Free dátových štruktúr

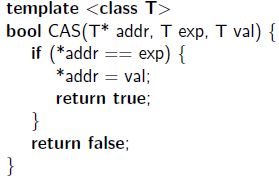
* Programovanie paralelných aplikácií bez použitia zamykania alebo iných mechanizmov
* Používanie ejdnej atomickej inštrukcie
* Minimálna latencia súvisejúca s prístupom dát
* Neexistuje uviaznutie, garantovaná životnosť
* Algoritmicky zložitejšie uvažovanie
* Korektnosť algoritmu náchylná na optimalizácií prekladača

## Wait-Free procedura

* Procedúra ktorá bez ohľadu na súber dvoch a viac vlákien dokončí svoju činnosť v konečnom čase, tj. Neexistuje súbeh ktorý by ju nutil nekonečne dlho čakať

## Lock-Free procedura

* Procedura, ktorá garantuje že pri ľubovoľnom súbehu mnoha súťažiacich vlákien vždy aspoň jendo úspešne skončí svoju činnosť
* Niektoré môžu byť ľubovoľne dlho nútené odkladať dokončenie svojej činnosti
* **Maurice Herlihy** – CAS konštrukcie sú vhodné na budovanie lock-free dat. Štruktúr
* Súčasné procesory majú HW podporu pre CAS



* Chybný scenár ak Vlákno A načíta hodnotu premennej a tá sa rovná x, dovtedy však iné vlákna zmenia hodnotu na rozličné od x, v okamžiku aplikovania inštrukcie CAS môže mať znova hodnotu x, vlákno A nepozná že sa hodnota objektu zmenila a CAS aj tak uspeje
* Odstránilo to réžiu súvisejúcu so zamykaním ale zostáva však réžia s koherencí cache pamätí
* Win32 **InterlockedCompareExchange()**

# Hazardné ukazatele

* Vlákna vystavujú ostatným vláknam zoznam ukazateľov, ktoré momentálne používajú
* Bezpečne ide dealokovať len objekty, ktoré nie sú odkazované žiadnym hazardným ukazovateľom

## Metoda Acquire()

* Pre zverejnenie používaného ukazateľa

## Metoda Release()

* Zneplatnenie objektu, tj. Oznámi že ukazateľ uložený v tomto objekte už nie je ďalej využívaný

## Metoda Retire()

* Nahradzuje funkciu delete
* Ak je zoznam ukazovateľov príliš dlhý volá procedúru Scan ktorá odstráni nepoužívané ukazatele

## Metoda Scan()

* Vytvorí zoznam používaných ukazatelov a zoradí ho
* Binárnym polením zisťuje či je ešte používaný
* Naďalej nepoužívané objkty dealokuje

# MCAS

* Rozšírenie štandartnej inštrukcie CAS pre použitie s ľubovoľne veľkou dátovou štruktúrou

# Transakčná pamäť

* Pamäť je modifikovaná v jednotlivých transakciách
* Transakcia zoskupuje mnoho čítaní a zápisov do pamäti
* Základným manipulovatelným objektom je slovo procesoru
* Príklad: presun prvku v dynamicky zreťazenom zozname

# Load-Link/Store-Conditional

* Dvojica inštrukcií, ktorá dokopy realizuje CAS
* LL načíta hodnotu a SC ju prepíše, ak nebola modifikovaná
* LL/SC rovnaká sila ako CAS avšak nemá ABA problém
* Problémy:
* zmena kontextu sa v praxi považuje za modifikáciu miesta
* Teoreticky nie je možné realizovat wait-free proceduru
* Obtiažne ladenie

# OpenMP

* Pragma direktivy prekladača

#pragma omp direktiva [zoznam klauzuli]

* Obsahuje knihovné funkcie, premenné prostredia
* Preklad kódu –fopenmp

### #pragma omp parallel

* Štrukturovaný blok sa prevedie paralelne
* Mimo sekvenčne
* Vlákno ktoré naň zastaví sa stane hlavným vláknom ID = 0

### Podmienené spustenie

* Klauzula **if(bool)**
* Ak false ignoruje sa paralel

### Stupeň paralelismu

* Počet vlákien
* Prednastavený počet ľpecifikovaný premennou prostredia
* Klauzula: **num\_threads(int)**

### Klauzula:

* **private**(zoznam premennych), premenne sa zduplikujú sa stanú sa lokálnymi premennými v kažom vlákne
* **firstprivat(**zoznam premenných**)**, rovnako ako private ale hodnotz premennych su inicializovane hodnotou originalnej kopie
* **shared(**zoznam premennych**),** vymenované premenné budu existovať iba v jednej kópií
* **default([shared|none])**
* shared: všetky premenné sú zdieľané, ak nie je uvedené inak
* none: vynucuje explicitné uvedenie každe premennej v klauzuli private alebo shared
* **reduction(**operátor: zoznam premenných**)**
* pri ukončení paralelného bloku sú privátne premenné spojené daným operátorom
* premenné musia byť skalárneho typu
* Operátory : + \* - & | ^ && ||

### Direktíva for:

* Iterácie nasledujúceho cyklu sú vykonané paralelne
* #pragma omp parallel for
* private, firstprivate ,reduction

### Klauzula lastprivate

* Hodnota privatnej premennej vo vlákne spracovávajúca poslednú iteráciu je uložená do kopie premennej platnej po skonceni cyklu

### Klauzula ordered

* Bloky označené ordered v tele cyklu sa vykonávajú v rovnakom poradí ako by boli vykonané sekvenčne

### Klauzula: nowait

* Jednotlivé vlákna sa nesynchronizujú po vykonaní cyklu

### Klauzula: schedule (typ planovania[,velkosť])

* Určuje ako budú iterácie rozdelené medzi vlákna

#### Static

* Iterácie bloku sú rozdelené blokov o špecifikovanej veľkosti
* Bloky staticky namapované na vlákna
* Ak nieje uvedená veľkosť, iterácie sa medzi vlákna rozdelia rovnomerne

#### Dynamic

* Bloky cyklov prideľované vláknam na žiadosť

#### Guided

* Bloky iterácii majú veľkosť proporcionálne k počtu nezpracovaných iterácii
* Premenná k špecifikuje minimálnu veľkosť bloku
* Príklad: k = 7, 200 voľných iterácii , 8 vlákien

200/8 = 25

#### Runtime

* Typ plánovania určený až za behu programu premennou OMP\_SCHEDULE

### Direktíva sections:

* Štrukturovaný blok označený sections obsahuje bloky označené section a tieto bloky sú vykonané paralelne
* #pragma omp parallel sections
* Umožňuje definovať rôzny kód ore rôzne vlákna

### Klauzuly : private, firstprivate, reduction, nowait

### Klauzula lastprivate: hodnoty privátnych premenných v poslednej sekcií budú platné po skončení bloku sections

### Direktíva barrier

* #pragma omp barrier, vzťahuje sa k najbližšej direktíve parallel
* Nutnosť ho uviesť v mieste kódu kde sa bezpodmienečne vykoná

### Direktíva single

* V kontexte paralélneho bloku sa blok označený single vykoná iba ejdným vláknom, jedno ktorým
* Klauzuly: private, firstprivate, nowait – ak nie je uvedená = na konci bloku single bude bariera

### Direktíva master

* Špeciálny prípad direktívy single, kedy kod označený master sa vykoná na hlavnom vlákne

### Direktíva critical

* Blok je chápaný ako KS a môže byť vykonaný max 1 vláknom v danom čase
* KS je pomenovaná a e možné iným vláknom vykonávať KS s iným názvom
* #pragma omp critical [(name)]

### Direktíva atomic

* Nahradzuje KS nad jednoduchými modifikáciami (update premennej)
* Aplikuje sa na jeden jednoduchý výraz (jeden load a store)
* Neatomizovatelný výraz : x = y = 0;

### Direktíva flush

* #pragma omp flush [(zoznam)]
* Explicitná direktíva pre kopírovanie hodnoty premennej z registrov do pamäti a späť
* Použitie: po zápise do zdielanej rpemennej, pred čítaním obsahu zd. Premennej

### Direktíva threadprivate a copyin

* Perzistentné privátne premenné – pretvajú zánik vlákna
* Znovupoužitie pri vytvorení nového vlákna, potrebný rovnaký počet vlákien, nepoužitelné pri dynamic plánovaní

#### Direktíva copyin

* Ako threadprivate ale s inicializáciou

## OpenMP knihovné funkcie

* void omp\_set\_num\_threads(int num\_threads)
* pred klauyulou parallel, prebité klauzulou num\_threads ak je prítomná, musí byť povolené OMP\_DYNAMIC (omp\_set\_dynamic())
* int omp\_get\_num\_threads()
* int omp\_get\_max\_threads() vracia maximálny počet vlákien v týme
* int omp\_get\_thread\_num() vracia ID vlákna vrámci týmu
* int omp\_get\_num\_procs() počet procesorov ktoré su dostupné na paralelný výpočet
* int omp\_in\_parallel() vracia nenula ak je volané vrámci paralelného bloku
* void omp\_set\_dynamic(int dynamic\_threads)
* int omp\_get\_dynamic()
* Nastavuje a vracia či je programátorovi dovolené dynamicky meniť počet vlákien
* Nenulová hondnota dynamic\_threads značí povolené
* void omp\_set\_nested(int nested)
* int omp\_get\_dynamic()
* Nastavuje a vracia či je povolený vnorený paralelizmus
* Ak nie, vnorené bloky sú serializované
* 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 alebo ničí, blokujúco čaká, odomyká a testuje – normálny a rekurzívny mutex

## Premenné prostredia

* OMP\_NUM\_THREADS, defaultný poče vlákien pri použití parallel
* OMP\_DYNAMIC hodnota TRUE umožnuje za behu meniť počet vlákien
* OMP\_NESTED hodnotou TRUE povoľuje vnorený paralelizmus, false vnorené budu serializované
* OMP\_SCHEDULE udáva defaultné nastavenie mapovania iterácií na vlákna
* „static,4“ , dynamic, guided

# Intel’s Thread Building Blocks (TBB)

* C++ knihovňa na vytváranie parallel apl.
* Založená na Generic Programming
* Vyvinuté spojením OpenMP direktív, STL šablón a jazyov podporujúcich rácu s vláknami
* Nepotrebuje podporu špeciálneho prekladača
* Podporuje nested parallelism
* Obsahuje šablóny pre:
* Paralelizáciu cyklov
* Definíciu vlastných paralelne dostupných dát. Štruktúr
* Využitie nízkourovňových HW primitív
* Zamykanie prístupu do KS v rôznych podobách
* Ľahkú definíciu paralelných súbežných úloh
* Škálovatelnú alokáciu pamäti

### Paralel for

* Dva parametre : rozsah a telo cyklu
* **template<typename Range,typename Body> void** **parallel\_for(const Range& range, const Body& Body)**
* Šablona zaistí vykonanie tela cyklu pre každý index
* Koncept delenia:
* Nový typ koštruktora – deliaci **X::X(X& x,split)**
* Rozdelí inštanciu triedy X na dve časti, dokopy dávajú pôvodný object
* Jedna časť je priradená do x druhá do novo vytvoreného objektu
* Split – špec. Trieda na odlíšenie deliaceho konštruktora od kopírovacieho
* Rozsah:
* Kopírovací konštruktor **R::R(const R&)**,deliaci **R::R(const R&,split)**, destructor **R::~R()**, test na prazdnost **bool R::empty() const**, test na schopnost dalsieho delenia **bool R::is\_divisible() const**
* **blocked\_range, blocked\_range2d**

### Blocked range

* template<typename value> class blocked\_range;
* Reprezentuje nadalej delitelny interval **[i,j)**
* Trieda **Value**
* Kopírovací konš. , Deštruktor, Operátor porovnania, operátor mínus,operátor plus
* Konštruktor špecifikuje interval rozsahu a veľkosť najväčšieho ďalej nedeliteľného interval
* **blocked\_range<int>(**Value begin, Value end [,size grainsize]**)**
* **blocked\_range<int>(5,17,2)**

### Parallel\_reduce

* **template<typename Range, typename Body>**

**void parallel reduce( const Range& range, const Body& body);**

* **Body::join(Body& to\_be\_joined)**  realizuje redukciu hodnôt zrôznych rozsahov
* **Body::operator()(Range&** range**)** realizuje redukciu nad daným rozsahom

## Trieda Partitioner

* Paralélna koštrukcia, tetí volitelný parameter špecifikuje stratégiu delenia rozsahu
* **parallel\_for<Range,Body,Partitioner>**

### Stratégie

* simple\_partitioner, rekurzívne rozdelí intervaly až na dalej nerozdeliteľné
* auto\_partitioner, automatické rozdelenie, zohľadňuje zaťaženie vlákien

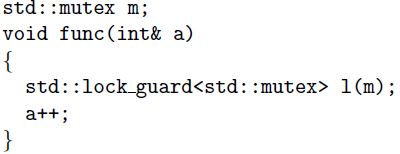
## Paralélne kontajnery

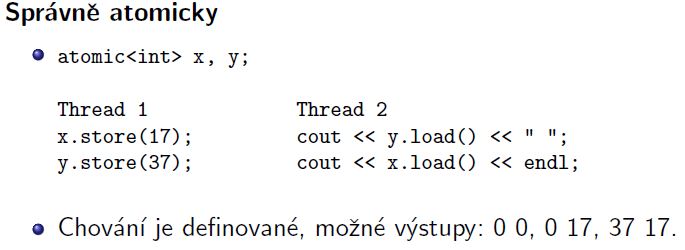
* concurrent\_queue
* veľkosť fronty je daná počtom operácií vloženia bez počtu operácií výberu, záporná hodnota značí čakajúce výbery
* concurrent\_vector
* zväčšovateľné pole prvkov, možné súbežný prístup z viac vlákien a zároveň súbežné zväčšovanie poľa
* concurrent\_hash\_map
* **template<typename Key, typename T, typename HashCompare>**

**class concurrent\_hash\_map;**

* mapa v ktorej je možné paralélne hľadať, mazať a zväčšovať
* HashCompare – obsahuje metodu (bool)**equal** a (size\_t)**hash**
* Obsahuje accessor pre prístup v režime read/write a const\_accessor pre read only
* Ich použitie umožňuje korektný prístup k zdieľaným premenným
* bool find(const accessor& result, const Key& key) const
* bool find(accessor& result, const Key& key)
* bool insert(const accessor& result, const Key& key)
* bool erase(const Key& key)

# C++11

* Trieda **lock\_guard**
* Obalenie štandartného zámku
* Mute nejde predať inému vláknu
* Trieda **unique\_lock**
* Všeobecnejšie predateľné obalenie mutexu, aj pre použitie s podmienkovými premennými



# Práca naviac pre programátora paralel. aplikácii

* Identifikovať súbežne proveditelné činnosti
* Mapovať súbežne vykonateľné časti práce do procesov
* Zaistiť distribúciu vstupných, vnútorných a výstupných dát
* Spravovať súbežný prístup k zdieľaným premenným
* Synchronizovať jednotlivé procesy v rôznych štádiách výpočtu

# Návrh a realizácia paralelného systému

# 

## Dekompózícia

* Proces rozdelenia celej výpočetnej úlohy na podúlohy
* Niektoré podúlohy môžu byť vykonané paralélne

### Podúlohy

* Jednotky výpočtu získanie dekompozíciou
* Po vyčlenení sa považujú za dalej nedelitelné
* Sú definované v dobe kompilácie/ za behu

## Graf závislostí

* Zachytáva závislosti vykonávaných úloh
* Definuje relatívne poradie vykonávaných úloh
* Orientovaný acyklický graf
* Môže byť nespojitý či dokonca prázdny
* Napríklad: poradie obliekania zvrškov oblečenia Topologické usporiadanie

## Granularita

* Počet úloh, na ktoré sa problém dekomponuje
* Mnoho malých úloh – jemnozrnná granularita fine-grained
* Málo väčších – hrubozrnná gr. Coarse-grained

## Stupeň súbežnosti

* Maximálny počet úloh ktoré môžu byť vykonané paralelne

## Priemerný stupeň súbežnosti

* Závislý na grafe závislostí a granularite
* Majme množstvo práce asociované k uzlom grafu
* Kritická cesta – cesta , na ktorej je súčet práce maximálny
* Priemerný stupeň súbežnosti – podiel celkového množstva práce voči množstvu práce na Kritickej ceste
* Udáva maximálne zrýchlenie pokiaľ je cieľová platforma schopná vykonávať súbežne maximálny stupeň súbežnosti úloh
* Čím menej práce na Krit. ceste tým väčší potenciál na paralelizáciu

## Interakcia úloh

* Nezávislé úlohy môžu vzájomne komunikovať

# Techniky dekompozície

* Všebecné dekompozície: Rekurzívna vs Dátová
* Špecializované dekompozície: Prieskumová,Špekulatívna, Hybridná

## Rekurzívna dekompozícia

* Vhodné pre problémy typu **Rozdeluj a panuj**
* Problémy sa dekomponujú na podúlohy tak, aby jednotlivé úlohy mohli byť dekompované rovnakým spôsobom ako rodičovská úloha
* Niekedy je treba reštrukturalizovať ulohu
* Hľadanie minima v lineárnom poli – polenie prehľadavaného poľa

## Dátová dekompozícia

* Data sa rozdelia na časti
* Úlohy sa vykonávajú súbežne nad jendotlivými časťami dát
* Podľa miesta:
* Vstupné data
* Výstupné data
* Vnútorné data
* Kombinácia
* Typické pre „Embarrassingly parallel“ – dekompozícia na dostatočný počet zcela nezávislých, vzájomne nekomunikujúcich úloh

## Prieskumová dekompozícia

* Vhodné pre prehliadavajúce úlohy, priehľadávaný priestor sa rozdelí podľa smeru hľadania
* Pri znalosti prehľadavaného priestoru ide dosiahnuť optimálneho vyváženia a zaťaženia CPU
* Úloha končí hned ako je nájdené požadované
* V prípade že Graf nie je strom treba riešiť problém opakujúcich sa konfigurácií

## Špekulatívna dekompozícia

* Úloha, ktorá čaká na vstup od predchádzajúcej úlohy sa spustí nad všetkými možnými vstupmi
* Vykonáva sa zbytočná práca
* Vo výsledku nemusí byť rýcheljšia ako serializovaná vezia
* Vhodné pre úlohy kde istá hodnota medzivýsledkov má veľkú pravdepodobnosť
* Príklad: Špekulatívne vykonávanie kódu

## Hybridná dekompozícia

* Kombinácia rôznych spôsobov dekompozície
* Hľadanie minima v poli
* sekvenčná verzia O(n), táto O(n/p + log(p))
* Vstupné pole sa rozdelí na p častí
* Najdú sa minimá v jednotlivých častiach v čase O(n/p)
* Výsledky jednotlivých zotriedení sa skombinujú v čase O(log(p))
* Teoretický pri dostatočnom množstve procesorov sa dá dosiahnuť O(log(n))
* Tento štýl paralelizmu sa volá **MAP-REDUCE**

# Techniky mapovania a vyrovnávania záťaže

## Mapovanie

* Priradzovanie úloh jednotlivým vláknam/procesom
* Optimálne mapovanie berie do úvahy graf závislostí a interakcie
* Ovplyvňuje výkon aplikácie
* Naivné mapovanie (úloha= proces/vlákno)
* Ciele mapovania:
* Minimalizovať celkový čas riešenia celej úlohy, redukovať idle time, záťaž,réžiu spustenia, ukončovania, prepínania,vyrovnávať záťaž na procesoroch
* Maximalizovať súbežnosť
* Minimalizovať zaťaženie dátových ciest
* Využitie dostupných zdrojov použitých predchádzajúcou úlohou
* Spôsob zadania úlohy:
* Statické zadanie – dekompozícia problému je zadaná v dobe kompilácie alebo odvodená od vstupných dát
* Dynamické zadanie – nové úlohy sú vytvárané za behu aplikácie podľa priebehu výpočtu
* Interakcie:
* Statické – priebiehajú v preddefinovanom časovom intervale
* Dynamické – ak dopredu nevieme počet interakcií
* Réžia spojená s mapovaním
* Prispôsobenie aplikácie s neočakávanou interakciou
* Pripravenosť dát k odoslaniu/adresáta k priajtiu
* Riadenie prístupu k zdieľaným zdrojom
* Optimalizácia aplikácie pre redukciu delay

# Schémy pre Statické mapovanie

### Mapovanie založené na rozdelení dát

* Bloková distribúcia
* Cyklická a blokovo-cyklická distribúcia
* Náhodná distribúcia blokov
* Delenie grafu

## Mapovanie založené na rozdelení úloh

* Delenie podľa grafu závislostí úloh
* Hierarchické delenie

# Mapovanie založené na rozdelení dát

## Bloková distribúcia

* Procesy zviazané s dátami rozdelenými na súvislé bloky
* Bloky môžu bť viacrozmerné
* Príklad: Násobenie matíc

## Cyklická a blokovo-cyklická distrubúcia dat. Polí

* Nerovnomerné množstvo práce spojené s jednotlivými prvkami
* Blokové delenie spôsobuje nerovnomerné zaťaženie
* Blokovo-cyklická distribúcia: delenie na menšie diely a cyklické priradenie procesom
* Zmenšovanie blokov vedie k cyklickej disribúcií

## Náhodná distribúcia blokov

* Záťaž súvisejúca s prvkami poľa vytvára pravidelné vzory
* Zlá distribúcia v cyklickom rozdelení
* Náhodné priradenie blokov procesom

## Grafové delenie

* Pre prípad kedy nie je vhodné organizovať data do polí
* Data organizované ako graf
* Optimálne delenie, rovnaký počet vrcholov v jednotlivých častiach,čo možno najmenší počet hrán medzi jednotlivými časťami

# Mapovanie založené na rozdelení dát

## Graf závislostí

* Grafové delenie
* Binárny strom => rekurzívna kodempozícia

## Hierarchické mapovanie

* Definuje hierarchie (vrstvy)
* Úlohové delenie, neberie na ohľad neuniformitu úloh
* Shlukovanie úloh do nadúloh

# Schémy pre Dynamické mapovanie

* Statické mapovanie nedostatočné pre úlohy, ktorých charakteristika nie je známa v dobe prekladu

## Centralizované schémy dynamického mapovania

* Úlohy sú zhromažďované v jendom mieste
* Dedikovaná úloha pre priradzovanie úloh procesorom
* Samo-plánovanie - akonáhle proces dokončí úlohu, vezme si ďalšiu
* Blokové plánovanie – prístup k zhromaždišťu úloh môže byť obtiažny => prideľovanie úloh po blokoch
* Príklad: Triedenie prvku v  **n** x **n** matici A

## Distribuované schéma

* Množina úloh je distribuovaná medzi procesy
* Za behu prebieha vzájomné vymieňanie úloh
* Netrpí nedostatkami spojenými s centralizovaným riešením
* Možnosti:
* Ako sa určí kto a komu pošle úlohu
* Kto a na základe čoho určí že úloha ma byť presunutá
* Koľko úloh má byť presunutých
* Kedy a ako je úloha presunutá
* Problém : Efektivita prenosu úlohy na iný proces

# Afinné plánovanie

* Modifikácia algoritmu plánovania
* Afinné plánovanie zaisťuje že výpočetné dávky jedného procesu/vlákna budú pridelené na ten istý fyzický procesor
* Výhody: lepšie využitie cache
* Nevýhody: môže narušovať vyváženosť a teda redukovať výkon aplikácie

# Metódy pre redukciu réžie interakcie

* Réžie spojené s interakciou súbežných úloh je kľúčovým faktorom ovplyvňujúcim výkon aplikácie
* Z pohľadu réžie sú ideálne „Emarrassingly parallel“ úlohy kde nedochádza k interakcií
* Faktory ovplyvňujúce réžiu:
* Objem prenášaných dát
* Frekvencia interakcie
* Cena komunikácie
* Cieľ
* Presun zdieľaných dát. Štruktúr do lokálnych kópií
* Minimalizácia objemu zdieľaných dát
* Lokálne ukladanie medzivýsledkov
* Viac správ v jednom
* Priestorová lokalizácia prenášaných dát
* **Problém:**
* prístup k obmädzenému zdroju v ten istý okamžik je serializovaný, to spôsobuje problémy
* čakanie na príjem či odoslanie dát spôsobuje nechcené prodlevy
* opakovane drahé prístupy k zdieľaným zdrojom => kópia dát
* Rovnaká interakcie medzi procesmi vykonávaná základnými komunikačnými primitívami je velmi drahá
* Nedostatočná priepustnosť komunikačej siete

# Redukcia ceny komunikácie

* Komunikácia v nezdieľanom adresnom priestore
* Synchronizácia a predávanie dát pomocou posielania správ
* Komunikácia v zdieľanom adresovom priestore
* Synchronizácia korektným prístupom k zdieľaným dátam
* Predávanie dát pomocou zdieľaných dát. štruktúr (FIFO)

## Všeobecné charakteristiky

### Latencia

* Doba potrebná k doručeniu prvého bitu
* Celková cena - Ts - pre zahájenie komunikácie (čas pre prípravu dát,identifikácia adresáta,+ doba vyliatia informácie z cache do pamäti alebo siete)
* Cena „hopov“ - Th – čas strávený na routroch v sieti, doba po ktorú putuje hlavička správy z prijímacieho na odosielací port

### Prenosová rychlosť

* Ovplyvnené šírkou pásma r
* Cena za prenos jedného slova 2 bajty – Tw = 1/r

### Cena komunikácie

* M – dĺžka správy v slovách
* L – počet liniek, cez ktoré správa putuje
* Ts + L \* (Th+ m\*Tw)

## Metódy redukcie ceny

* Spojovanie malých správ zmenšuje sa hodnota Ts
* Kompresia – znižovanie hodnoty m
* Minimalizácia vzdialenosti – znižovanie vzdialenosti (L)
* Paketovanie – eliminácia réžie spôsobené jednotlivými skokmi
* Ts + L \* (Th+ m\*Tw) => Ts + LTh + mTw

# Kolektívne komunikačné primitíva

* Komunikácia (interakcia) – predávanie informácií medzi jednotlivými procesmi

## Parametre komunikácie

* Latencia – Ts oneskorenie spojené so začatím vlastnej komunikácie
* Šírka pásma – Tw maximálne množstvo dát prenesených za jednu jednotku času
* Objem – m množstvo predávaných správ
* Cena: **Ts + m \* Tw**

# Tolopógia komunikačnej siete

* Fyzická/Logická štruktúra komunikačných kanálov medzi jednotlivými účastníkmi komunikácie

## Vlastnosti komunikačnej siete

* Priemer (dĺžka maximálnej najkratšej cesty)
* Konektivita – minimálny počet disjunktívnych ciest
* Stupeň – maximálny počet liniek ktoré prichádzajú do styku s ejdným vrcholom
* Cena
* Výlučnosť prístupu – použitie jednou úlohou blokuje ostatných
* Rozšíriteľnosť
* Škálovateľnosť

## Zbernica

* Zdieľané médium
* Priepustnosť klesá s počtom uzlov

## Kliky

* Privátne neblokujúce spojenie každého s každým
* Cena: počet liniek je vkadratický voči N
* Cena: stupeň každého uzlu je N-1
* Škálovatelné za podmienky lacného zvyšovania stupňa uzlu

## Prsteň

* Usporiadanie na uzloch
* Privátne neblokujúca komunikáca s 2 najbližšími uzlami

## Hviezdica

* Spojenie cez jeden centrálny uzol
* Stredový uzol môže byť úzkym miestom
* Dá sa hierarchicky vrstviť

## Hyperkocka

* Spojenie n uzlov v tvare log2n rozmernej kocky
* Stupeň uzlu log2n

## Strom s aktívnymi vnútornými uzlami

* Strom kde participianti komunikácie sú listy i vnútorné uzly

# OTA

* Úloha posiela niekoľkým/všetkým identické dáta
* Vo výsledku je p kópií pôvodnej správy
* Pre ring topológiu je sieť nevyužitá (komunikujú stále len susedné uzly)
* Technika rekurzívneho zdvojenia
* Polenie intervalov prvý adresát je p/2,p/4,p/8...
* OTA v 2D sieti
* Sqrt(p) reťazí o sqrt(p) uzloch
* Najprv správy do každej reťaze a potom po reťazi
* Cena: 2\*(log(sqrt(p)))
* OTA v d-rozmernej sieti
* Rovnaký princíp d\*(log())

# ATO

* Duálna operácia k OTA
* Niekoľko/Všetky úlohy posielajú dáta jednej úlohe
* Dáta sa kombinujú pomocou zvoleného operátoru
* ATO prsteň
* Procesy s nepárnym číslom zašlú správy procesom s ID o 1 menším, správy sa skombinujú
* Následne sa info presunú na pozície ktoré sú násobkom 4
* ...

## Univerzálne algoritmy OTA a ATO

* Predpoklad 2d uzlov (hyperkocka)
* Každý uzol identifikovaný bitovým vektorom
* Cena : d(Ts + mTw)

# ATA vysielanie

* All to All
* Každá úloha posiela rovnakú informáciu všetkým uzlom
* Vo Výsledku je p kópií p originálnych správ
* P \* m => P \* P \* m
* ATA prsteň
* Každý pošle info susedovi => potom zbierajú a preposielajú dalej, všetky linky využité
* ATA reťaz
* Vysielanie správ susedom na obe strany Full-duplex (n-1 fáz),Half-duplex 2(n-1)fáz

# ATA redukcia

* Všetci posielajú informáciu všetkým, prichádzajúce správy sa kombinujú
* Každá úloha má pre každú inú úlohu inú správu
* P \* P \* m => P \* m
* ATA prsteň / reťaz
* ATA mriežka/hyperkocka po každej faze objem správ je zredukovaný na polovicu
* Správy po jednej posielané po kruhu tak že správa pre najvzdialenejší je poslaná ako prvá,pri priechode uzlom sa prikombinuje správa danáho uzla k prechádzajcej správe

# ATA individuálna komunikácia

* Každá úloha posiela rôzne dáta ostatným úlohám
* Dojde k 2D výmene správ  **p** x **p** v 1D priestore p
* „Totálna výmena“
* Príklad: transpozícia matice
* ATA individual Prsten, retaz
* Ulohy pošlú jedným smerom p-1 úloh v každom ďalšom kole si každá úloha vyextrahuje správu ktorá je jej urcčená a ostatné prepošle dalej
* ATA individual mriežka
* Medzi retiazkami a potom v retiazkach
* ATA individual hyperkocka nie je to optimálny algoritmus

# E-cube routing

* Cesta emdzi 2 bodmi je daná pozíciou odlišných bitov týchto bodov, rotuje sa od najmenej význameného bitu
* Vyžaduje duplexné linky

# Prefixový súčet

* Úlohy zasielajú dáta ostatným úlohám s rovnakým či menším ID
* Dáta sa kombinujú
* [3,1,4,0,2] => [3,4,8,8,10]

# Scatter – Scan

* Jedna úloha posiela unikátnu správu každej ďalšej úlohe
* One-To-All individual personalizovaná komunikácia
* Žiadne dáta sa neduplikujú

# Gather – Concatenation

* Jedna úloha zbiera unikátne dáta od ostatných úloh
* All-To-One individual personalizovaná komunikácia
* Nevyskytuje sa tu kombinácia dát

# Cyklický posun

* Permutácia – všeobecná komunikačná primitíva, jedna úloha posiela dáta jednej inej úlohe, v 1D rotaciou o q pozícii v 2D Akcelerácia s využitím druhej dimenzie

# Send a Receive

* send(void \*sendbuf, int nelems, int dest)
* receive(void \*recvbuf, int nelemns, int src)
* sendbuf – ukazatel na bafer (blok pamäti), kde su umiestene data pripravene k odoslani
* recvbuf – ukazatel na bafer (blok pamäti), kam budu umistene prijate data
* nelems – počet dátovych jednotiek, ktore budu poslane, či přijate (dlžka správy).
* dest – adresat odosielanej spravy, tj. *ID* toho, komu je sprava určena
* src – odosilatel spravy, tj. *ID* toho, kto spravu poslal, alebo toho, od koho chcem spravu prijať
* Operácia send() je ukončená až je to bezpečné, jej ukončenie negarantuje ani nevynucuje že ju prijemca dostal
* Operácia receive končí až po prijatí a zapísaní dát na správne meisto v pamäti
* Pred samotným prenosom prebieha handshake
* Obmädzená velkosť buffer => môže vzniknúť buffer overflow

# Message Passing Interface MPI

* Štandartizuje syntax a semantiku komunikačných primitív
* MPI\_Init, MPI\_Finalize,MPI\_Send,MPI\_Send,MPI\_Recv
* MPI\_Init nastavuje prostredie, musí byť volané ako prvá MPI funkcia,argv nesmie byt modifikovane MPI\_Init(int \*argc,char\*\*\* argv), musí byť volaný na každom CPU
* MPI\_Finalize() ukončuje prostredie, nesmie za ním byť volaná žiadna MPI funkcia, musí byť volané na vš. Procesoroch
* Komunikačné domény : združovanie partipujúcich CPU do jednej skupiny, môžu sa prekrývať
* Komunikátory : premenné ktoré uchovávajú komunikač. Domény
* int MPI\_Comm\_size(MPI\_Comm comm, int \*size)
* int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)
* rank je identifikator procesu v danej domene, čislo v intervale [0,size-1]
* int MPI\_Send(void \*buf, int count, MPI\_Datatype datatype,

int dest,int tag, MPI\_Comm comm)

* Odošle data na pointeri buf
* Data interpretované ako sekvencia inštancií typu datatype
* Odošle sa count iterácií po sebe iúcich inštancií
* Dest je rank adresáta v danej doméne
* Tag je informácia int v interval [0,MPI\_TAG\_UB], odlišuje typ správ
* int MPI\_Recv(void \*buf, int count,MPI\_Datatype datatype, int source,int tag,

MPI\_Comm comm,MPI\_Status \*status)

* Prijme spravu od odosielatela s rankom **source** v komunikačnej domene **comm**

s tagom tag

* Ak je buffer malý odosle sa error MPI\_ERR\_TRUNCATE
* **tag** môže byť MPI\_ANY\_TAG
* MPI\_Sendrecv súčasné posielanie a prijímanie správ

## Neblokujúce Send a Receive

* MPI\_Isend,MPI\_Irecv
* MPI\_Test test na dokončenie operácie
* MPI\_Wait blokujúce čakanie na dokončenie operácie
* MPI\_Request\_free explicitné zničenie objektu request
* MPI\_Barrier
* MPI\_Bcast OTA
* MPI\_Reduce
* MPI\_Scan prefixova redukcia
* MPI\_Gather,MPI\_Scatter,MPI\_Alltoall

# Réžia (overhead) paralelných programov

## N-násobné zrýchlenie

* S použitím n-násobných HW zdrojom sa dá očakávať n-násobné zrýchelnie
* Málokedy nastáva z dôvodu réžie paralel. riešenia, ak áno dôvodom je neoptimálne riešenie sekvenčného algoritmu
* Dôvody: réžia – interakcia,prestoje(synchronizácia,čakanie na zdroje/dáta), složitosť paralel. alg.

# Metriky výkonnosti

## Čas výpočtu – sekvenčný algoritmus

* Doba, ktorá uplynie od spustenia výpočtu do jeho ukončenia
* **Ts**

## Čas výpočtu – paralélny algoritmus

* Doba, ktorá uplynie od spustenia výpočtu do doby kedy skončí aj posledný z paralélnych výpočtov
* **Tp**

## Celková réžia

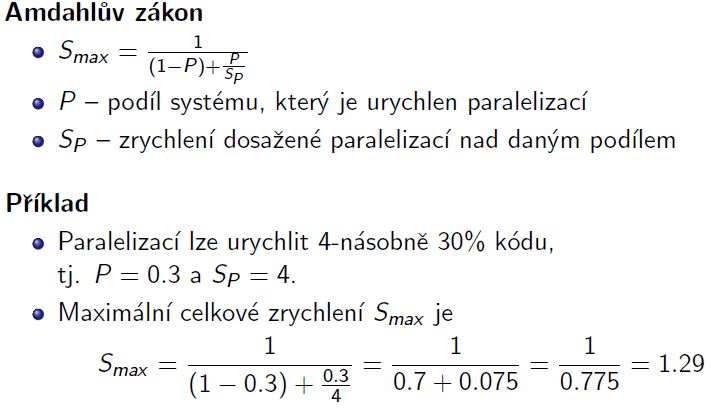
* **To**
* Všetky elementy kré spôsobujú réžiu paralel. riešenia
* Celkový čas paralel. výpočtu bez času potrebného pre výpočet sekvenčným riešením
* Paralélne a sekvenčné algoritmy môžu byť úplne odlišné

## Funkcia celkovej réžie

* Doba od skončenia výpočtu jedného procesu do skončenia celého paralélneho výpočtu sa považuje za Réžiu (Idling)
* **To = pTp – Ts**

## Miera účinnosti paralelizácie

* Pomer času výpočtu na jednej CPU jednotke a na p procesných jednotkách
* **S = Ts / Tp**
* S použitím **p** procesných jednotiek je maximálne zrýchlenie **p**
* **Super-lineárne zrýchelnie** – jav kedy je zrýchlenie väčšie ako p

****

## Efektivita

* Podiel zrýchlenia voči počtu jednotiek **p**
* **E = S / p = (Ts/Tp)/p = Ts/ (p \* Tp)**
* Zrýchlenie je v praxi < p a Efektivita v rozmedzí (0,1]
* „Podil času po ktorý jednotka vykonáva užitočný kód“

## Cena riešenia problému

* Súčin počtu CPU jednotiek a doby paralélneho riešenia : **C = p \* Tp**
* Označované tiež ako „pTp“ produkt

## Deškálovanie scale-down

* Umelo znižujeme granularitu -> hrubšie úlohy
* Snižujeme overhead
* Môže ovplyvniť cenu a cenovu optimalitu

## Škálovateľnosť

* Zachovanie výkonu a efektivity pri zvyšovaní počtu procesných jednotiek
* Efektivita sa dá zachvoať pri súbežnom zvyšovaní **Ts** a **p**