

1. Paralelní RAM model a ošetření konfliktů při přístupu do sdílené paměti

Paralelní RAM model:

Množina p procesorů, každý procesor P_i má lokální paměť a index i .

Sdílená paměť se skládá z m paměťových buněk $M[j]$

Každý procesor P_i může přistoupit do jakékoliv buňky $M[j]$ v konstantním $O(1)$ čase.

Vstup PRAM algoritmu: n položek v *(typicky prvních)* n buňkách sdílené paměti

Výstup PRAM algoritmu: n' položek v n' buňkách sdílené paměti

Operace PRAM modelu:

- čtení buňky sdílené paměti (READ – R)
- zápis do buňky sdílené paměti (WRITE – W)
- operace v lokální paměti (LOCAL – L)

Modely: jednotkový (každá operace trvá čas 1) / globální (L – čas 1, R/W – čas $d > 1$)

Ošetření konfliktů při přístupu do sdílené paměti:

- EREW: exkluzivní čtení i zápis (*ke sdílené paměti přistupuje pouze 1 procesor*)
- CREW: současné čtení více procesory, exkluzivní zápis pouze 1 procesorem
- CRCW: současné čtení i zápis, nutnost řešení současného zápisu:
 - a) priority: každý procesor má svou prioritu, zápis může dokončit jen s nejvyšší prioritou
 - b) arbitrary: dokončení zápisu je povoleno náhodnému procesoru
 - c) common: všechny procesory mohou dokončit zápis jen v případě, že zapisují stejná data – jinak není stav počítače definován.

2. APRAM model a implementace synchronizační bariéry

Asynchronní PRAM model:

Procesory pracují asynchronně, provádí operace READ, WRITE, LOCAL stejně jako v PRAM. Doba přístupu do sdílené paměti **není** jednotková, nutnost explicitní synchronizace.

APRAM výpočet: posloupnost globálních fází, kde procesory pracují asynchronně a synchronizací (*dva procesory **nemůžou** přistupovat do stejné buňky sdílené paměti ve stejné globální fázi, pokud alespoň jeden zapisuje*)

Výkonnostní parametry APRAM modelu:

- lokální operace: čas 1
- globální operace READ nebo WRITE: čas d
- k po sobě jdoucích globálních operací: $d + k - 1$
- bariérová synchronizace: $B(p)$

Implementace bariéry – centrální čítač (lineární, $B(p) = O(dp)$)

- začíná iniciovaný na 0 v příchozí fázi, procesy přistupují k paměti exkluzivně (jednotlivě)
- 1. proces dorazí v bariéře, zkontroluje příchozí fázi a inkrementuje čítač
- 2. pokud je čítač $< p$, deaktivuje se, jinak bariéru nastaví do odchozí fáze a vše aktivuje
- 3. poslední aktivovaný proces nastaví bariéru do příchozí fáze

Implementace bariéry – binární redukční strom (logaritmický, $B(p) = O(d \cdot \log(p))$)

1. každý proces dorazí k bariéře a zkontroluje, zda je v příchozí fázi
2. počká, než skončí redukce v jeho podstromu a po jejím skončení pošle signál rodiči
3. kořen stromu čeká na redukci z obou podstromů, pak přepne do odchozí fáze
4. procesy se zaktivují ve zpětném pořadí (*od kořenu k listům*)

3. Paralelní čas, zrychlení, cena, efektivnost a paralelní optimalita výkonnosti

Paralelní čas $T(n, p)$:

- čas od začátku paralelního výpočtu do chvíle, kdy poslední procesor skončí výpočet
- počet paralelních výpočetních kroků + počet komunikačních kroků (*závisí na architektuře*)
- závisí na architektuře paralelního počítače (*třeba brát v úvahu při hodnocení*)
- n = velikost problému, p = počet vláken výpočtu

$SU(n)$: horní mez nejrychlejšího existujícího sekvenčního algoritmu pro zadaný problém

Paralelní zrychlení $S(n, p) = \frac{SU(n)}{T(n, p)}$

Lineární zrychlení: $S(n, p) = \Theta(p)$

- při běhu na p vláknech je čas p krát menší
- v reálu velmi obtížně dosažitelné (*závisí na paralelizovatelnosti dílčích výpočtů*)

Superlineární zrychlení: $S(n, p) > p$

- může nastat v situaci, kdy má např. sekvenční algoritmus velkou paměťovou složitost

Paralelní cena: $C(n, p) = p \cdot T(n, p)$

- důležitá metrika, jelikož většina paralelních architektur alokuje výpočetní jádra staticky
- **optimální cena:** $C(n, p) = \Theta(SU(n))$

Paralelní efektivnost: $E(n, p) = \frac{SU(n)}{C(n, p)}$

- relativní vytížení výpočetních zdrojů během paralelního výpočtu
- **optimální:** 1, kvůli komunikační a synchronizační režii ale bude běžně menší než 100 %
- **konstantní efektivnost:** pokud dokážeme najít $0 < E_0 < 1$ takové, že $E(n, p) \geq E_0$

Paralelní optimalita: paralelní algoritmus je optimální, právě když

- ⇔ je cenově optimální
- ⇔ má lineární zrychlení
- ⇔ má konstantní efektivnost

4. Amdahlův zákon, Gustafsonův zákon, izoeфекtivní funkce

Amdahlův zákon: každý sekvenční algoritmus A se skládá z

- inherentně sekvenčního podílu f_s , který může provést jen 1 vlákno ($0 < f_s < 1$)
- paralelizovatelného podílu $1 - f_s$

Pokud provedeme paralelizaci algoritmu A pro pevné n pomocí $p > 1$ vláken, pak ideálně:

$$S(n, p) = \frac{1}{f_s + \frac{1 - f_s}{p}} \leq \frac{1}{f_s}$$

Neboli nezávisle na počtu spuštěných vláken nemůže zrychlení přesáhnout $1/f_s$.

Důsledek: pro každý algoritmus existuje p , kdy se už nevyplatí přidávat další procesory, protože pro ně už není dostatek paralelní práce.

Gustafsonův zákon: s rostoucím p máme úměrně navyšovat i velikost problému n

- inherentně sekvenční část trvá vždy konstantní čas t_{seq} nezávisle na p (vstup, výstup)
- inherentně paralelní část t_{par} lineárně škáluje s p v čase

Paralelní škálovatelnost: schopnost paralelního počítače se zvětšit, pokud narůstá velikost řešeného problému – **silná** (měří pokles efektivnosti při rostoucím p a konstantním n – Amdahlův zákon) a **slabá** (měří růst n při rostoucím p a konstantní efektivnosti – Gustafson)

Izoeфекtivní funkce ψ_1, ψ_2 :

- $\forall n_p = \Omega(\psi_1(p)) : E(n_p, p) \geq E_0$ – dolní mez velikosti problému v závislosti na počtu procesorů za účelem udržení konstantní efektivnosti
- $\forall n_p = O(\psi_2(n)) : E(n, p_n) \geq E_0$ – horní mez počtu procesorů v závislosti na velikosti problému za účelem udržení konstantní efektivnosti

Z Amdahlova zákona vyplývá: $p = \omega(\psi_2(n))$ – abychom udrželi konstantní efektivnost, musí být procesorů alespoň $\psi_2(n)$.

Z Gustafsonova zákona vyplývá, že když velikost problému roste s p vztahem $n = \Omega(\psi_1(p))$, efektivnost nebude klesat.

5. OpenMP: programový model, paralelní region, vlastnosti proměnných

OpenMP:

- explicitní model paralelního výpočtu = programátor má plnou kontrolu nad paralelním výpočtem a nese za něj zodpovědnost
- vysokoúrovňové API, virtuálně sdílená paměť, datový i funkční model paralelismu

Paralelní region: *#pragma omp parallel*

- část kódu, ve kterém jsou pomocí **fork-join** mechanismu vytvářena, prováděna a ukončována paralelní vlákna
- mimo paralelní region existuje pouze 1 hlavní **master** vlákno
- z regionu se nesmí skákat ven nebo dovnitř nebo provádět thread unsafe operace

Vlastnosti proměnných v paralelním regionu:

- **shared**: proměnná je sdílena všemi vlákny
- **private**: proměnná je lokální pro každé vlákno, není inicializována
- **firstprivate**: proměnná je lokální, inicializuje se na původní hodnotu v master vlákně
- **lastprivate** (*v paralelních cyklech*): proměnná je lokální, ale po skončení poslední iterace se přepokopíruje její obsah do proměnné hlavního vlákna procesu
- **threadprivate** (*na daném vlákně, musí být více regionů za sebou se stejným počtem vláken*)
- **default**: jakou z předchozích vlastností budou mít implicitně všechny proměnné v regionu

6. OpenMP: datový paralelismus (direktiva for), sémantika, parametry

Datový (iterační) paralelismus:

- přidělujeme n iterací cyklu for p vláknům
- na konci je implicitní bariéra (*pokračuje se, až všechny iterace skončí*)

Klauzule:

- schedule(typ): určuje způsob přidělení iterací cyklu vláknům
- collapse(): umožňuje paralelizaci vnořených cyklů (*ve výchozím nastavení ne*)
- ordered: pořadí provádění iterací je stejné jako při sekvenčním provádění
- nowait: vlákna po dokončení iterací cyklu neprovádějí bariéru

Typy klauzule schedule direktivy for:

- schedule(static, chunk-size = n/p): každému vláknům je staticky přidělen blok po sobě jdoucích iterací o zadané velikosti (*výchozí velikost n/p*)
- schedule(dynamic, chunk-size=1): vláknům jsou dynamicky přidělovány bloky po sobě jdoucích iterací o zadané velikosti (*výchozí velikost je 1*)
- schedule(guided, chunk-size=1): vláknům jsou dynamicky přidělovány bloky x iterací, kde $x = \max(\text{dosud nepřidělených iterací} / p, \text{chunk-size})$
- schedule(runtime): způsob je určen až v okamžiku spuštění pomocí systémové proměnné OMP_SCHEDULE
- schedule(auto): způsob je ponechán na kompilátoru nebo běhovém prostředí

7. OpenMP: funkční paralelismus (direktiva task), sémantika, parametry

Funkční paralelismus:

- vytváříme jednotlivé úlohy – tasky (direktiva task), ty se přidají do task poolu
- zde si úlohu vyzvedne první volné vlákno a začne ji provádět
- mechanismus přidělování typu producent – konzument (*vlákna jsou obojí*)

Úloha:

- ukazatel na začátek kódu, který se má provést
- vstupní data
- datová struktura, do které vloží svůj identifikátor vlákno, které kód začne provádět

Podmíněné spuštění paralelních úloh:

- klauzule taskif = úloha se vytvoří, jen pokud například počet úloh není větší než x
- příklad: `#pragma omp task taskif if(pos < len / TASK_LEN)`

8. OpenMP: synchronizační direktivy

Všechny základní mechanismy synchronizace přístupu vláken do sdílené paměti.

Barrier: na dané místo musí dorazit a počkat všechna vlákna paralelního regionu

Master: daný blok kódu smí provést pouze **hlavní** vlákno

Single: daný blok může provést libovolné **jedno** vlákno

Critical: vytvoření kritické sekce – umožní výlučný přístup ke sdíleným prostředkům

Atomic: daná paměťová operace nad paměťovou buňkou (*Read / Update / Write / Capture*) obsahující skalární datový typ (*int, float, double*) bude provedena atomicky – na jednom vlákně a nepřerušitelně

Flush: propsání aktuálních hodnot sdílených proměnných do sdílené paměti

Taskwait: synchronizace synovských úloh s rodičovskou v **task** paralelismu

9. Paralelní algoritmy pro prohledávání stavového prostoru, anomálie prohledávání, statické rozdělení prostoru a dynamické vyvažování zátěže

Prohledávání kombinatorického stavového prostoru (PKSP):

- NP-těžká úloha, hledáme **jeden** konkrétní stav v celém prostoru
- vstupní / stavové / výstupní proměnné, podmínky, omezení, optimalizační kritéria
- nezajímají nás heuristiky, simulované ochlazování, genetické programování
- provedeme **paralelizaci** hrubou silou

Paralelní algoritmy:

- BFS: generuje stavy ve stejné hloubce najednou
- DFS: generuje nezávislé cesty směrem ke koncovým stavům

Branch & bound DFS: provedu návrat i z mezistavu, který nemůže dát lepší řešení, než je aktuální dolní mez

DFS s postupným prohlubováním: hledám a postupně zvětšuji hloubku, ve které hledám

Statické rozdělení stavového prostoru (pomocí BFS):

- hlavní vlákno expanduje strom do p podstromů, každému vláknu přidělí 1
- vlákna předají výsledek zpět hlavnímu vláknu, které zkonstruuje globální řešení

Anomálie statického prohledávání:

- může se stát, že 1 vlákno dostane malý podstrom, zatímco jiné vlákno velký podstrom
- ostatní vlákna pak musí čekat na dokončení práce na vláknu s největším podstromem
- může dojít k **superlineárnímu** zrychlení, nebo **zpomalení**

Dynamické rozdělení:

- pokud vlákno dokončí práci (pomocí DFS), stane se nečinným, ale hledá si další práci
- a) některé vlákno se s ním rozdělí si s ním polovinu zbývajících práce
- b) master-slave: hlavní vlákno (*master*) vygeneruje podprostory, práci rozděljuje master

Tady možná říct ten finální workflow ze semestrální úlohy: pomocí BFS rozdělím práci na jednotlivé vlákna, ta pak pomocí DFS provádí prohledávání, když jim dojde práce, vezmou si další – OpenMP: `parallel for`, MPI: čekám na další práci pomocí `MPI_recv`.

10. Klasifikace paralelizovatelných OpenMP programů a zdroje jejich neefektivity, falešné sdílení a jeho eliminace

Klasifikace paralelizovatelných programů:

- a) výpočetně intenzivní algoritmy: čas trávený výpočtem nad daty > čas přesunu dat
 - NP-těžké úlohy, faktorizace matic, násobení matic => **jdou** škálovat a paralelizovat
- b) paměťově intenzivní algoritmy: čas strávený výpočtem < čas nutný na přesun dat
 - lineární výpočetní složitost, ale nutnost přesunu velkého množství dat
 - skalární součin, dynamické programování, Fourierovy transformace
 - výkonnost není dána výpočetní kapacitou, ale rychlostí paměti

Optimalizace sekvenčních kódů:

- snaha maximalizovat počet výpočetních operací na jeden načtený byte + využití cache
- načítání celých bloků cache, vyhnout se nepřímé adresaci

Zdroje neefektivity OpenMP programů:

- nevyvážená výpočetní zátěž (*kvůli bariéře se čeká na nejpomalejší vlákno*)
- příliš těsná synchronizace (*hodně bariér a kritických sekcí – hodně času na synchronizaci*)
- omezený paralelismus (*méně iterací nebo tasků než vláken*)
- vysoká režie správy vláken (*hodně malých tasků*)
- velká sekvenční část (*Amdahlův zákon*)
- neefektivní práce s keší (*častý zápis do sdílených proměnných, falešné sdílení*)

Falešné sdílení:

- situace, kdy procesory sdílejí jen cache a můžou si ji navzájem zneplatňovat
- nastává u datového paralelismu, vede k výpadku cachí a zpomalení výpočtu
- „více vláken zapisuje do jednoho pole, které je v jednom cache bloku“

Eliminace falešného sdílení:

- rozdělení řešeného problému tak, aby každý procesor dostal celé bloky cache paměti
- alternativně: umělé nafouknutí řešených dat podle velikosti bloku cache (*zvětší paměťovou náročnost, proto je lepší ho nepoužívat*)

11. Paralelizace výpočtu histogramu v OpenMP

Histogram: graf četnosti výskytu hodnot vstupního pole velikosti n , rozsah hodnot $range$

Standardní algoritmus – $O(n) + O(range)$:

- na začátku inicializace `for (i in 0..range) result[i] = 0`
- `for (i in 0..n) result[data[i]] += 1`
- každou hodnotu načte jednou a zahodí, nepřímá indexace => nevhodné pro paralelizaci

Paralelní řešení (naivně) – $O(range) + O(n/p)$:

- použití `#pragma omp parallel for` s `#pragma omp atomic update`
- bude velmi **neefektivní** – výpadky cachí (nepřímá indexace), velká synchronizace

Paralelní řešení (chytře) – $O(range) + O(n/p)$:

- rozdělení dat na p částí, spočtení dílčích histogramů
- následná **paralelní redukce** výsledků z jednotlivých lokálních histogramů
- **efektivní** – nedochází k výpadkům cachí

By default počítám u `parallel for` s blokovým rozdělením = `schedule(static)`.

12. Paralelizace násobení polynomů v OpenMP

Problém: máme dva pole koeficientů polynomů o velikosti m a n , chceme výstupní polynom

Sekvenční algoritmus $O(n * m)$:

- na začátku opět inicializace `for (i in 0..n+m) result[i] = 0`
- `for(i in 0..n) for (j in 0..m) result[i + j] += a[i] * b[j]`
- opět nevhodné pro paralelizaci kvůli nepřímé indexaci

Paralelní algoritmus – vnější cyklus (i): $O(n * m / p)$

- použití `#pragma omp parallel for` na vnějším cyklu s `atomic update`
- opět dochází k nepřímé indexaci a výpadkům cache => nebude efektivní

Paralelní algoritmus – vnitřní cyklus (j): $O(n * m / p)$

- použití `#pragma omp parallel for` na vnitřním cyklu
- zpomalení kvůli synchronizace bariér při každé další iteraci vnějšího cyklu
- už nebude docházet k nepřímé indexaci, ale může dojít k falešnému sdílení

Paralelní algoritmus nad výstupem: $O(n * m / p)$

- každé vlákno počítá část výstupních koeficientů, na konci paralelní redukce
- např. x^2 může vzniknout jako $x^0 * x^2$, $x^1 * x^1$ nebo $x^2 * x^0$
- musíme vyvážit zátěž (*dynamicky / staticky s vhodným chunk size*)
- dá se vyhnout falešnému sdílení, pokud vhodně rozdělíme podle velikosti bloků cache

13. Paralelizace násobení hustých matic (MMM)

Násobení hustých matic: máme dvě matice A, B o rozměrech $n \times n$, výsledek: matice C

Sekvenční algoritmus $O(n^3)$:

- klasický školní algoritmus, pro každý index výsledné matice provedeme skalární součin
- `for(i in 0..n) for(j in 0..n) c=0 for(k in 0..n) c += A[i][k] * B[k][j] result[i][j] = c`

Paralelizace vnějšího cyklu (i): $O(n^3 / p)$

- použití `#pragma omp parallel for` na vnějším cyklu
- procesory se střídají v počítání jednotlivých řádků – zapisují do disjunktních oblastí
- minimální synchronizace (*jen 1 bariéra na konci regionu*), nedochází k falešnému sdílení

Paralelizace vnitřního cyklu (j): $O(n^3 / p)$

- použití `#pragma omp parallel for schedule(static)` na vnitřním cyklu s j
- procesory se střídají v počítání jednotlivých bloků – zapisují do disjunktních oblastí
- větší synchronizace (*n bariér*), při dostatečně velkém n/p nedojde k falešnému sdílení
- pokud bychom rozdělili data cyklicky (`schedule(static, 1)`), došlo by k falešnému sdílení

Paralelizace vnitřního cyklu (k): $O(n^3 / p)$

- použití `#pragma omp parallel for` na vnitřním cyklu s k
- procesory pomocí paralelní redukce počítají jednotlivé hodnoty ve skalárním součinu
- velká synchronizace (*n^2 bariér*), pouze 1 vlákno zapisuje do výsledku – bez kolizí cache

14. Formáty pro uložení řídkých matic a paralelizace násobení řídké matice vektorem (MVM) v OpenMP

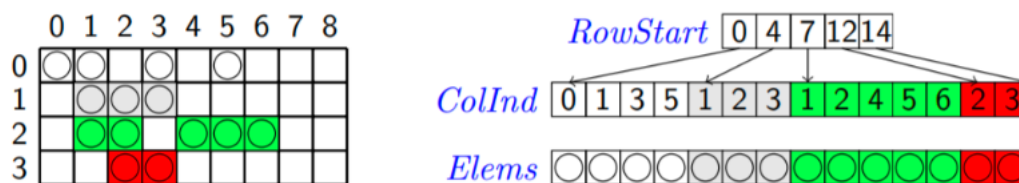
Řídké matice: více nul než ne-nul – ukládám v paměti pouze jako pole souřadnic ne-nul
Počet prvků matice: n

Formáty pro uložení řídké matice – souřadnicový (COO):

- držím si tři pole – pole řádků (row_index), pole sloupců (col_index), pole hodnot (element)
- pro každé k od $0..n$: $result[row_index[k]][col_index[k]] = element[k]$

Formáty pro uložení – komprimované řádky (CSR):

- opět tři pole – řádků (row_start), sloupců (col_index) a hodnot (elements)
- tentokrát ale pole řádků obsahuje indexy do pole col_index, kde začíná daný řádek



Násobení řídké matice vektorem: mám matici A, vstupní vektor x a výstupní vektor y

Sekvenční algoritmus: $O(n)$

- COO: `for (k in 0..n) y[row_index[k]] += elements[k] * x[col_index[k]]`
- CSR: `for (row in 0..row_count) sum=0`
 `for (k in row_start[row] .. row_start[row + 1])`
 `sum += elements[k] * x[col_index[k]]`
 `y[row] = sum`

Paralelní algoritmus (COO) – $O(n/p)$:

- použití `#pragma omp parallel for` na vnějším cyklu s `atomic update`
- paralelizace není efektivní – dochází ke kolizím a výpadkům cache

Paralelní algoritmus (CSR) – $O(n/p)$:

- použití `#pragma omp parallel for schedule(static)` na vnějším cyklu
- nedochází ke kolizím, pokud je dostatečně velké n , nedochází ani k výpadkům cache
- pokud nemá matice rovnoměrně zaplněné řádky, bude program zpomalen, jelikož budou mít jednotlivá vlákna **různou** výpočetní zátěž

Paralelní algoritmus (CSR) – cyklické nebo dynamické rozdělení:

- cyklické rozdělení: bude docházet k falešnému sdílení
- dynamické rozdělení: větší režie, ale pokud je vhodná velikost bloku, nedojde k falešn.sd.

Paralelní algoritmus (CSR) – vyvažování:

- pokud mám nerovnoměrně zaplněné řádky, můžu si je „sloučit“ do pruhů (*bands*)
- pak paralelizuji for cyklus nad pruhy místo řádků
- stále nedochází ke kolizím a výpadkům cache, ale i pro různě zaplněné řádky **stejně rychlé**

15. Základní myšlenky paralelizace QuickSortu a paralelního rozdělení v OpenMP

QuickSort $O(n \log(n))$: vybereme pivot, rozdělíme na levou/pravou část a rekurzivně seřadíme

Základní task paralelizace $O(n \log(n) / p)$:

- provedeme pomocí direktivy `#pragma omp task` na levou a pravou část
- pomalé – velká režie vytváření tasků, čekání na dokončení podvláken

Vylepšení task paralelizace:

- prah počtu tasků pomocí `#pragma omp task if(depth < DEPTH_THRESHOLD)`
- nahrazení jednoho ze dvou volání iterací (*sníží počet tasků na polovinu*)
- vyvažování částí (*podle velikosti levého a pravého pole přerozdělí i počet vláken*)
- paralelizace sekvenčního rozdělení (*bude tedy paralelní řazení i rozdělování*)

Paralelizace sekvenčního rozdělení:

- máme pivot, pole, potřebujeme ho prohodit tak, aby byly vlevo menší a vpravo větší prvky
- proces prohazování = **neutralizace** jednotlivých prvků

Hoareho paralelní algoritmus rozdělení:

- mám dva proti sobě jdoucí ukazatele – upravuji pomocí `#pragma omp atomic capture`
- levý větší než pivot, pravý menší než pivot => prohodím prvky, posunu ukazatele
- levý i pravý větší => posunu pravý ukazatel doleva || levý i pravý menší => levý doprava
- na konci zbydou „špinavá čísla“ => sekvenčně přehodím

Lepší paralelní algoritmus rozdělení:

- každé vlákno zpracovává bloky a porovnává je s pivotem, podle toho je zařadí za sebe
- myšlenky s ukazateli je stále stejná, akorát se po každém prvku nemění vlákno (*až po bloku*)
- na konci zbydou „špinavé bloky“ (*obsahuje prvky menší i větší*) => sekvenčně doseřadím

16. Základní myšlenky paralelizace MergeSortu a paralelního dvoucestného sloučení v OpenMP

MergeSort: rozdělím pole na dvě menší, ta seřadím, následně slučuji dvě seřazená pole

Základní task paralelizace:

- provedeme pomocí direktivy `#pragma omp task` na levou a pravou část, pak `taskwait`
- dochází k falešnému sdílení, rozdělení na mnoho malých úloh s režii => **nepoužitelné**

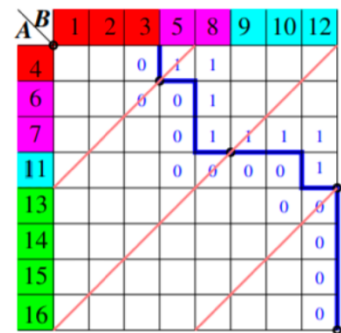
Vylepšení task paralelizace:

- prah počtu tasků pomocí `#pragma omp task if(depth < DEPTH_THRESHOLD)`
- nahrazení jednoho ze dvou volání iterací (*sníží počet tasků na polovinu*)
- paralelizace slučovací operace (*bude tedy paralelní řazení i spojování*)

Slučování: mám dvě seřazená pole A a B, chci je sloučit do výsledného pole C

Dvoucestné paralelní slučování:

- sekvenčně si vytvořím matici X: $X[i][j] = A[i] > B[j] ? 0 : 1$
- jelikož jsou pole seřazená, bude existovat přechod mezi 0/1
- matici proložíme $p-1$ diagonálami tak, aby vzniklo p bloků (*prokládáme diagonály rovnoměrně, nezávisle na datech*)
- pro každou diagonálu spočteme průsečík s přechodem (*binární půlení, $O(\log n)$*) – podle těchto průsečíků rozdělím
- každé vlákno pak sekvenčně sloučí bloky A_i a B_i
- na konci stačí zřetězit sloučené posloupnosti od každého vlákna
- pořád může docházet k falešnému sdílení, rychlejší je **p-cestné paralelní slučování**

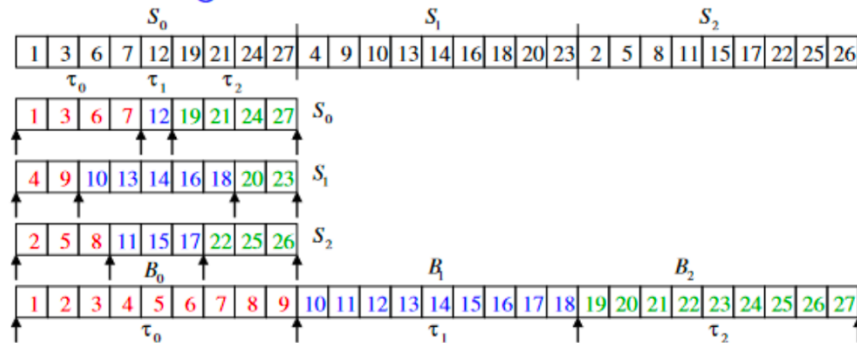


17. Základní myšlenky paralelního p-cestného MergeSortu v OpenMP

p-cestný MergeSort:

- rozdělím pole S na p menších $S[0], S[1], \dots, S[p-1]$ o velikosti n/p
- každé vlákno i seřadí příslušné pole $S[i]$
- následně si každé vlákno i kromě prvního spočítá v každé části $S[j]$ oddělovač tak, aby všechny jeho příslušné části měly dohromady n/p prvků
- jednotlivé úseky podle oddělovačů sloučím za sebe do výsledného pole B

Paralelní MergeSort: Příklad 3-cestného PMWMS



Po seřazení jednotlivých částí a výpočtu oddělovačů musí být **bariéry**.

Výpočet oddělovačů:

- vlákno vezme sdílené pole p se seřazenými podčástmi $S[0], S[1], \dots, S[p-1]$
- spočítá pozici oddělovače a index, kde začíná jeho úsek ve výstupním poli

Asymptotický odhad paralelního času:

$$T(n, p) = O\left(\frac{n}{p} \cdot \log\left(\frac{n}{p}\right) + p \cdot \log\left(\frac{n}{p}\right) \log(n) + \frac{n}{p} \cdot \log(p)\right)$$

Části:

- sekvenční řazení n/p čísel
- $\log n$ provedení p hledání v polích velikosti n/p
- sekvenční p -cestné slučování p polí celkové délky n/p

Ve výsledku prakticky: n je velké, p je malé \Rightarrow bude dominovat první člen

18. OpenMP + MPI – spolupráce procesů a vláken

MPI: message passing interface

- standardizovaný a přenositelný systém zasílání zpráv, standard pro distribuované aplikace
- umožňuje **distribuované** programování = program běží na více uzlech, řídí MPI **knihovna**
- procesy si navzájem posílají zprávy (*rozdíl oproti OpenMP, kde procesy na 1 stroji*)
- pouze **knihovní funkce** (v OpenMP také direktivy)

Skupiny procesů: každý MPI proces je součástí skupiny procesů, procesy indexovány od 0

- MPI_Comm_rank: zjistí číslo v rámci skupiny
- MPI_Comm_size: zjistí počet procesů v dané skupině
- MPI_COMM_WORLD: výchozí skupina všech procesů MPI programu

Překlad programu:

- OpenMP: g++ -fopenmp main.cpp
- MPI: **mpicc++** main.cpp, nastavení konkrétního překladače: OMPI_CXX=g++

Paralelní redukce:

- OpenMP: #pragma omp parallel reduction(+:var_name)
- MPI: MPI_Allreduce(MPI_IN_PLACE, &var_name, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD)

Spolupráce OpenMP a MPI:

- funkce MPI_Init_thread udávající míru spolupráce
- SINGLE: na každém uzlu pouze jedno vlákno (*v podstatě pouze MPI*)
- FUNNELED: na uzlu více vláken, ale pouze master vlákno může volat MPI funkce (*1port*)
- SERIALIZED: na uzlu více vláken, ale volání MPI funkcí je kritická sekce – v 1 okamžiku může jen jedno vlákno může volat MPI funkce (*1 portový model*)
- MULTIPLE: na uzlu více vláken, každé vlákno může volat MPI funkce kdykoliv (*vše portový*)

19. MPI: blokující komunikační operace a jejich komunikační módy, stavové objekty

Blokující komunikační operace:

- po zavolání funkce je vlákno uspáno, dokud nedosáhneme určitého stavu (*výsledku*)
- **dvoubodové**: mezi dvěma procesy, **kolektivní**: mezi všemi procesy v daném komunikátoru

MPI_Send (const void *buffer, int size, MPI_Datatype, int dest, int tag, MPI_Comm communicator):

- odešleme data uložená v ukazateli buffer o velikosti size na proces s rankem dest
- datatype: typ posílaných dat, předdefinované výchozí MPI_INT, MPI_UINT32_T nebo možnost zadefinovat si vlastní datový typ MPI_Type_create_struct
- tag: odliší zprávy různého významu, můžeme použít číslo, které si vybereme

MPI_Recv (const void *buffer, int size, MPI_Datatype, int source, int tag, MPI_Comm communicator, MPI_Status *status):

- přijme maximálně size dat do ukazatele buffer od procesu s rankem source
- datatype, tag, communicator stejný význam

MPI_TAG_ANY: přijme zprávu s libovolným tagem

MPI_ANY_SOURCE: přijme zprávu od libovolného procesu v zadaném komunikátoru

Stavový objekt MPI_Status:

- obsahuje číslo zdrojového procesu (MPI_SOURCE) a značku přijaté zprávy (MPI_TAG)
- MPI_Get_count(status, MPI_Datatype, &count) zjistí počet přijatých prvků

Standardní mód MPI_Send:

- začne nezávisle na iniciaci příjmu dat, ukončí se až po:
 - a) předání dat cílovému procesu (*nelokální operace – závisí na přijetí cílem*)
 - b) překopírování dat do dočasného bufferu pro pozdější odeslání cílovému procesu (*lokální operace – i když cílový proces neinicioval přijetí dat, pokračuje se dál*)
- MPI rozhodne, která možnost nastane, nemůžeme klást **žádné předpoklady**

Buffered mód MPI_Bsend (lokální operace):

- začne nezávisle na tom, zda cíl inicioval přijetí dat, vždy se data předají do bufferu
- dost velký buffer musí uživatel připravit pomocí MPI_Buffer_attach (*jinak chyba*)
- dvojnásobná paměťová náročnost (*data ve dvou bufferech*), ale nemusíme čekat

Synchronous mód MPI_Ssend (nelokální operace):

- může začít, i když cíl neinicioval příjem dat, vrátí se až poté, co cíl iniciuje příjem dat
- vysílač i přijímač na sebe navzájem čekají

Ready mód MPI_Rsend (nelokální operace):

- může začít, pouze když cíl již inicioval příjem dat, pak pokračuje stejně jako synchronní mód
- pokud v době volání není cíl schopen přijmout data, skončí s chybou

20. MPI: neblokující komunikační operace a způsoby zjištění jejich dokončení, stavové objekty

Neblokující komunikační operace:

- lokální operace = iniciují odeslání / přijetí dat a hned se vrátí
- **nemůžeme** pracovat s bufferem, dokud se odesílání / přijetí nedokončí
- dokončení můžeme otestovat pomocí funkcí `MPI_Test` nebo `MPI_Wait`

Sémantika funkcí je totožná se sémantikou funkcí `MPI_Send` a `MPI_Recv`

Funkce: `MPI_Isend`, `MPI_Ibsend`, `MPI_Issend`, `MPI_Irsend`, `MPI_Irecv`

- `MPI_Irsend` může začít, až když příjemce iniciuje příjem, ostatní libovolně

Dokončení neblokujících operací – `MPI_Test`, `MPI_Wait`:

- operace `MPI_Irecv` nemá jako poslední parametr `MPI_Status`, ale `MPI_Request`
- `MPI_Test(MPI_Request *request, MPI_Status *status)`
- `MPI_Wait(MPI_Request *request, MPI_Status *status)`
- možnost čekat na libovolnou / všechny operace z pole typu `MPI_Request`:
`MPI_Testany` / `MPI_Waitany` / `MPI_Testall` / `MPI_Waitall`

Rozdíl:

- `MPI_Test` je neblokující, vrátí okamžitě hodnotu `MPI_SUCCESS` nebo chybu
- `MPI_Wait` je blokující, vrátí se až tehdy, když jsou data skutečně obdržena

21. MPI: sondování příchodu zprávy

Sondování příchodu zprávy: zjišťujeme, zda existuje zpráva, kterou lze přijmout

MPI_IProbe(int source, int tag, MPI_Comm, int *flag, MPI_Status *)

- lokální funkce, neblokující, nastaví flag na **true** pokud existuje zpráva, kterou lze přijmout s odpovídajícími parametry source, tag, communicator
- do argumentu status pak uloží stejnou hodnotu, jako kdyby bylo voláno MPI_Recv
- jedná se pouze o sondu = zpráva nemusí být přijata, může ji sondovat pak i někdo jiný

MPI_Probe(int source, int tag, MPI_Comm, MPI_Status *)

- nelokální funkce, blokující, vrátí až tehdy, kdy existuje zpráva, kterou lze přijmout

Požadavky na implementaci sondovacích funkcí:

- volání MPI_Probe by mělo úspěšně vrátit, pokud v té době zavolá jiný proces MPI_Send
- vrácení proběhne úspěšně, pokud jiné vlákno téhož procesu nevolalo Receive / Probe
- ve vícevláknových procesech může dojít k **soupeření** vláken o přijetí zpráv

MPI_Improbe(int source, int tag, MPI_Comm, int *flag,
MPI_Message * MPI_Status *):

- v případě úspěšného sondování, vrátí Improbe navíc **message handle** na zjištěnou zprávu

MPI_Mrecv(buf, MPI_Count, MPI_Datatype, MPI_Message *, MPI_Status *):

- **matching receive**, vyzvedne vysondovanou zprávu pomocí MPI_Improbe

22. MPI: návratové hodnoty funkcí a ošetření chyb

Indikace chyb v programu:

- příčinou může být chyba v programu, v hardwaru, nebo při komunikaci (*v síti*)
- MPI **neřeší** chyby v **komunikační infrastruktuře**, počítá se spolehlivou komunikací
- většina MPI funkcí vrací hodnotu MPI_SUCCESS při úspěchu nebo **kód** chyby jinak

Ošetření chyb:

- implicitně se při výskytu chyby nejprve zavolá error handler podle komunikátoru
- výchozí error handler pro MPI_COMM_WORLD: MPI_ERRORS_ARE_FATAL

Obsluha chyb (*error handler*):

- MPI_ERRORS_ARE_FATAL: násilně ukončí **celý** MPI program – pokud tedy zvolíme tuto možnost, nemusíme ani návratovou hodnotu kontrolovat
- MPI_ERRORS_RETURN: chyba program neukončí, ale vrátí kód chyby MPI funkce, stav výpočtu ale **není definován** – typicky chceme vypsát chybové hlášení a skončit
- MPI_ERRORS_ABORT: ukončí **jen** všechny procesy spojené s daným komunikátorem

Nastavení:

- MPI_Comm_set_errhandler(MPI_Comm, int error_handler)
- vlastní: MPI_Comm_create_errhandler, MPI_errhandler_free

23. MPI: Implementace permutace cyklický posuv

Permutace cyklický posuv: proces num posílá data procesu $(\text{num} + 1) \% \text{num_procs}$

Nefunkční řešení:

- každý proces zavolá MPI_Send dalšímu procesu a následně MPI_Recv od předchozího
- nebude správně fungovat, pokud MPI zvolí u některých procesů synchronní mód

Funkční řešení:

- každý lichý proces zavolá MPI_Send sudým procesům, sudé procesory MPI_Recv
- v druhém kroku sudé procesory pošlou data lichým
- není optimální, potřebujeme MPI_Bsend a buffer, ale problémem je velikost bufferu

Optimální řešení:

```
MPI_Sendrecv(const void * sendbuff, int sendcount,
              MPI_Datatype sendtype, int dest, int sendtag,
              void *recvbuf, int recvcount,
              MPI_Datatype recvtype, int source, int recvtag,
              MPI_Comm comm, MPI_Status *status)
```

Každý proces se chová jako **dvouportový** = dokáže přijmout data zleva a současně vyslat doprava, nejlepší a nejjednodušší řešení.

24. Základní grafové vlastnosti propojovacích topologií (řídke/husté topologie, hierarchicky rekurzivní topologie, vzdálenosti, regularita, souvislost, bisekční šířka, bipartitnost)

Topologie G_n : nekonečná množina instancí jednoho typu grafu s parametrem n (dimenze)
Inkrementálně / částečně škálovatelná topologie: definována pro všechna / některá n

Řídká topologie: stupně uzlů jsou omezeny konstantou $|E(G_n)| = O(|V(G_n)|)$

Hustá topologie: stupně uzlů jsou rostoucí funkcí n : $|E(G_n)| = \omega(|V(G_n)|)$

Hierarchicky rekurzivní topologie: instance **menších** dimenzí jsou podgrafy instancí **větších** dimenzí (příklad: n -rozměrná mřížka je složena kartézským součinem menších mřížek)

Vzdálenost uzlu u a v : délka nejkratší cesty mezi u a v

Excentricita uzlu u : nejdelší vzdálenost mezi uzlem u a libovolným uzlem v v grafu

Průměr grafu $diam(G)$: největší excentricita libovolného uzlu (největší vzdálenost)

Poloměr grafu $r(G)$: nejmenší excentricita libovolného uzlu (nejkratší z nejdelších cest)

Regulární graf: stupeň každého uzlu (počet sousedů) je roven nějaké konstantě k

Uzlový (hranový) řez $\kappa(G)$: množina uzlů (hran), jejichž odebráním se rozpojí souvislý graf

Uzlová (hranová) souvislost $\lambda(G)$: velikost minimálního uzlového (hranového) řezu
= minimální počet uzlů / hran, jejichž odebráním zruším souvislost grafu

Platí: pro libovolný graf **uzlová souvislost \leq hranová souvislost \leq minimální stupeň v grafu**
Rovnost nastává při **optimální** souvislosti.

(pokud má uzel stupeň 2, tak odebráním těch 2 hran už to rozpojím, někdy stačí i méně)

Bisekční šířka $bw_e(G)$: velikost nejmenšího hranového řezu grafu na **2 poloviny**

(kolik hran musím odebrat, abych dostal dvě přibližně stejné poloviny)

Bipartitní graf: graf, kde existuje obarvení vrcholů dvěma barvami tak, že koncové vrcholy každé hrany mají odlišnou barvu (rozdělím na dvě poloviny, hrany jen mezi nimi)

Optimální topologie:

- konstantní stupeň uzlu (umožní univerzální, levné směrovače a řídkou topologii)
- malý průměr a malá průměrná vzdálenost (snižuje komunikační zpoždění)

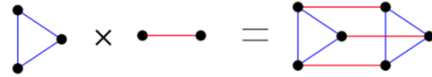
Spodní mez průměru optimální topologie: $\Omega(\log N)$

(každá řídká topologie s logaritmickým průměrem je optimální)

Požadavky na topologie:

- hierarchická rekurzivita: umožní snadnější škálovatelnost
- bisekční šířka: vhodná pro paralelní binární rozdělovačové algoritmy
- vysoká souvislost: redundantní cesty v případě výpadků nebo přetížení uzlů / linek, možnost rozdělit velké zprávy po paralelních disjunktních cestách

25. Kartézský součin grafů, uzlová symetrie



Kartézský součin: $G = G_1 \times G_2$

- $V(G) = \{[x, y]; x \in V(G_1), y \in V(G_2)\}$ (vezmu kartézský součin množin vrcholů)
- $E(G) = \{< [x_1, y], [x_2, y] >; < x_1, x_2 > \in E(G_1)\} \cup \{< [x, y_1], [x, y_2] >; < y_1, y_2 > \in E(G_2)\}$

(Hrana vede tehdy, pokud vedla původně v jednom nebo druhém grafu)

- **komutativní** a **asociativní** operace (nezáleží na pořadí a uzávorkování)

Uzlová symetrie:

- pro každé dva vrcholy existuje automorfismus (prosté, na) f takový, že $f(u_1) = u_2$
(Jsem schopný na sebe namapovat libovolné dva vrcholy, je jedno, z jakého úhlu se dívám)

Věta: každý graf vzniklý kartézským součinem uzlově symetrických grafů je uzlově symetrický

(Idea důkazu: pokud byly původní uzlově symetrické, existují automorfismy f_1, f_2 , které mi mapují na sebe uzly v G_1, G_2 . Pokud provedu kartézský součin, hrana vede pouze pokud vedla v původním G_1 nebo G_2 , takže to namapování bude sedět)

Platí: každý uzlově symetrický graf je **regulární** (jinak by mi neseseděly stupně uzlů)

Vzdálenosti v uzlově symetrickém grafu: průměr je stejný jako poloměr, jelikož všechny uzly mají stejnou excentricitu (vychází z vlastností automorfismu)

Optimální topologie a uzlová symetrie:

- umožní snazší návrh paralelních a komunikačních algoritmů – je jedno, odkud začnu

26. n-rozměrná hyperkrychle: definice, vlastnosti, směřování

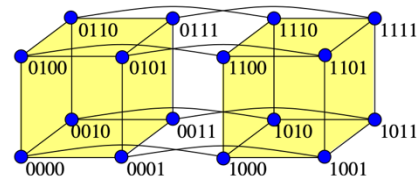
Ortogonalní topologie:

- sestavena kartézským součinem cest délky 1

Vrcholy: $\{0, 1\}^n$ – binární n-tice

Hrana: vede mezi vrcholy, pokud se liší **právě** v jednom bitu (0000 \rightarrow 0001, 0010, 0100, 1000)

Q_4 :



Počet vrcholů: $N = 2^n$

Počet hran: $n \cdot 2^{n-1}$ ($Q_{n-1} \times P_2$)

Průměr: n (nejdále jsou například 0000 a 1111)

Stupeň: n (n negací = n hran)

Bisekční šířka: 2^{n-1} (chci rozseknout na dvě $n-1$ krychle, ty mají 2^{n-1} vrcholů ke spojení)

Vlastnosti:

- regulární graf (vše stupeň n)
- hustá topologie (logaritmický stupeň uzlů – $n = \log(2^n)$)
- optimální souvislost (uzlová i hranová souvislost je rovna stupni uzlu – n)
- bisekční šířka je největší možná ($2^{n-1} = 2^n/2 = N/2$)
- vyvážený bipartitní graf (skládá se ze dvou podkrychlí, ty stačí obarvit)
- hamiltonovský graf (každý n -bitový Grayův kód je hamiltonovská kružnice Q_n)

Hamiltonovský graf: lze projít tak, že je každý vrchol navštíven právě jednou

Grayův kód: kód, který v každém kroku invertuje jeden bit, „zrcadlový kód“

(0000 \rightarrow 0001 \rightarrow 0011 \rightarrow 0010 \rightarrow 0110 \rightarrow 0111 \rightarrow 0101 \rightarrow 0100 ...)

- uzlově symetrická (kartézský součin $P_2 = Q_1$, která je uzlově symetrická)
- existuje $2^n \cdot n!$ různých automorfismů, $k!$ různých nejkratších cest ve vzdálenosti k
- počet uzlů ve vzdálenosti i od zadaného uzlu: $\binom{n}{i}$ – průměrná vzdálenost je horní celá část

Směřování: minimální e-cube směřování: zprava doleva (podle dimenzí)

Využití:

- není řídký graf, škálovatelné jen po mocninách 2 (částečně škálovatelná)
- základní testovací topologie, maximálně použita jako podsít

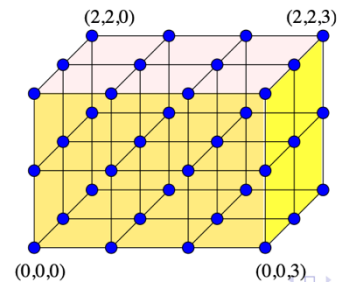
27. n-rozměrná mřížka: definice, vlastnosti, směřování

Ortogonalní topologie:

- sestavena kartézským součinem mřížek nižších $M(3, 3, 4)$:
dimenzí: $M(z_1, z_2, \dots, z_n) = M(z_1) \times \dots \times M(z_n)$

Vrcholy: (a_1, a_2, \dots, a_n) , kde a_i je vždy od 0 do příslušného z_n

Hrany: hrana vede mezi vrcholy, pokud se liší v právě jedné dimenzi o jedničku ($012 \rightarrow 022, 112, 002, 011$)



Počet vrcholů: $z_1 \cdot \dots \cdot z_n$ **Průměr:** $\sum_{i=1}^n (z_i - 1) = \Omega^n \sqrt{|V(M)|}$ (Z jednoho rohu do druhého)

Počet hran: $\sum_{i=1}^n (z_i - 1) \cdot \prod_{j=1, j \neq i}^n z_j$ **Stupeň uzlu:** $O(2 \cdot \sum_{i=1}^n z_i)$

(V každé dimenzi je $z_i - 1$ hran, a rozprostře se to po ploše rovné součinu ostatních dimenzí)

Bisekční šířka: $\Omega(N / \max(z_i))$ (chci říznout přes nejdelší hranu)

- v případě, že je délka nejdelší hrany sudá, bez Ω

Vlastnosti:

- není regulární graf (pokud z_i není stejné pro každé i) \Rightarrow není uzlově symetrická
- nejčastější 2-D (rovinné) a 3-D (kubické) mřížky
- **hierarchicky rekurzivní** (krychle dána součinem mřížek nižších dimenzí)
- optimální souvislost (uzlová i hranová souvislost rovna stupni uzlu)
- bipartitní graf (skládá se ze dvou podmřížek, ty stačí obarvit), ale ne nutně vyvážený
- hamiltonovský graf, pokud alespoň jedna hrana sudou délkou

Směřování: dimenzně uspořádané směřování (XY, XYZ směřování v 2D, 3D mřížkách)

28. n-rozměrný toroid: definice, vlastnosti, směřování

Ortogonalní topologie:

- sestaven kartézským součinem toroidů nižších dimenzí: $K(z_1, z_2, \dots, z_n) = K(z_1) \times \dots \times K(z_n)$

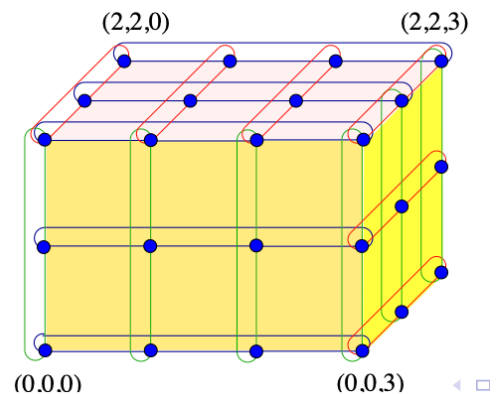
Vrcholy: (a_1, a_2, \dots, a_n) , kde a_i je vždy od 0 do z_n

Hrany: hrana vede mezi vrcholy, pokud se jejich

XOR liší v právě jedné dimenzi o jedničku

(012 -> 022, 112, 002, 011, **312, 010**)

(V podstatě mřížka, akorát spojujím i krajní hrany)



Počet vrcholů: $z_1 \cdot \dots \cdot z_n$ **Průměr:** $\sum_{i=1}^n \lfloor \frac{z_i}{2} \rfloor$ (je to vlastně kružnice = nejdál z rohu do středu)

Počet hran: $n \times \prod_{i=1}^n z_i$ (v každé dimenzi mám „součin“ počet hran, dimenzí je n)

Stupeň uzlu: $2n$

Bisekční šířka: $\Omega(2N / \max(z_i)) = 2bw_e(M(\dots))$

(jako mřížka, ale ještě musím seknout toroidové hrany)

Vlastnosti:

- regulární a uzlově symetrický graf (kartézský součin kružnic – ty jsou uzlově symetrické)
- průměr je **poloviční** vůči stejné velké mřížce, souvislost a bisekční šířka je **dvojnásobná**
- **hierarchicky rekurzivní** (na rozdíl od mřížek ale nelze rozložit na stejnorozměrné toroidy)
- hamiltonovský a vyvážený bipartitní graf

Směřování: podobně jako v mřížkách / hyperkrychách, postupně podle dimenzí (XY, XYZ)

Využití: neúspěšnější komerční topologie pro masivně paralelní počítače

Porovnání: hyperkrychle nejdražší, toroid uprostřed, mřížka nejlevnější

29. Řídké hyperkubické topologie a tlusté stromy: definice, vlastnosti, směrování

Řídké hyperkubické topologie:

- nejlepší možné topologie, **konstantní** stupeň a **logaritmický** průměr
- škálovatelné hůře než hyperkrychle ($N = n2^n$)
- bisekční šířka největší možná ($N / \log(N)$)
- základní topologie pro většinu paralelních algoritmů

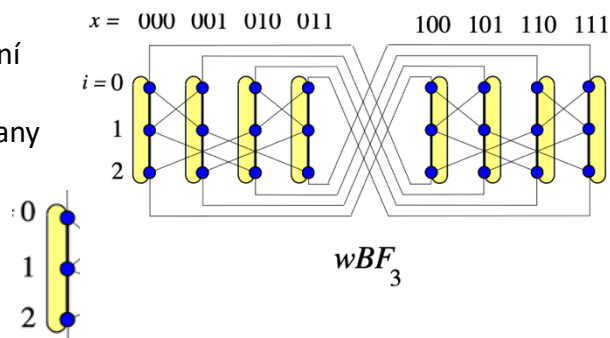
Zabalený motýlek wBF_n :

- není ortogonální ani hierarchicky rekurzivní
- vznikne **rozbalením** vrcholů hyperkrychle do kružnic – hyperkubické a motýlkové hrany

Motýlková hrana: v rámci kružnice \Rightarrow

Hyperkubická hrana: znegují i -tý bit, a vedu hranu do i o 1 menší/větší:

$$< (i, x), (i+1, \text{neg}_i(x)) >$$



Počet vrcholů: $n2^n$

Počet hran: $n2^{n+1}$ (počet vrcholů krát 2 – každý má stupeň 4)

Průměr: $n + \lfloor \frac{n}{2} \rfloor$ (musím přejít n hyperkubických hran, a pak $n/2$ motýlkových hran v kružnici)

Stupeň vrcholu: 4 (2 motýlkové, 2 hyperkubické hrany) **Bisekční šířka:** 2^n (rozseknu v půli)

Vlastnosti:

- řídký graf, **optimální** průměr, není hierarchicky rekurzivní
- hamiltonovský graf, bipartitní graf (vyvážený, pokud je n sudé)
- uzlově symetrický (můžu rotovat kružnici a přehazovat souřadnice, ale ne libovolně \Rightarrow pomocí XORu vhodně přeložím souřadnice)

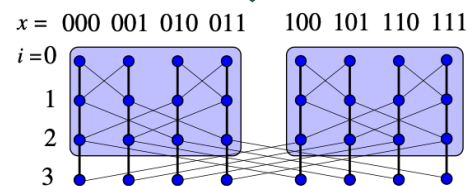
Obyčejný motýlek oBF_n :

- je hierarchicky rekurzivní (obsahuje dva motýlky dimenze $n-1$ jako podgrafy) a bipartitní
- vznikne **rozstřihnutím** zabaleného motýlka v 0 a přepojením hrany $<n-1, 0>$ do n

Přímá hrana: vždy mezi $<i, i+1>$ (rozstřihnutá kružnice)

Hyperkubická hrana: stejně jako u zabaleného mot.

(znegují i -tý bit, vedu hranu do i o 1 menší/větší)



Počet vrcholů: $(n+1)2^n$ **Počet hran:** $n2^{n+1}$

(stejný počet hran – jen jsme je přepojili, více vrcholů – 0 jsme „rozdvojili“)

Průměr: $2n = n + n$ (musím přejít n hyperkubických hran a pak n hran v kružnici)

Stupeň vrcholu: 2 nebo 4 (krajní $i=0, i=n-2$, ostatní 4) **Bisekční šířka:** 2^n (rozseknu v půli)

Vlastnosti: není uzlově symetrický a není regulární, není hamiltonovský, je **bipartitní**

Směrování: existuje pouze jedna nejkratší cesta mezi $(0, x)$ a (n, y) – e-cube směrování

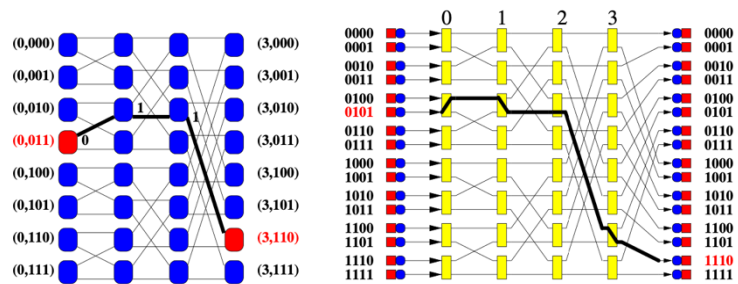
Využití: může sloužit jako minimální permutační síť = levná náhrada křížového přepínače

Normální hyperkubické algoritmy: v jednotlivých krocích používáme po sobě jdoucí dimenze

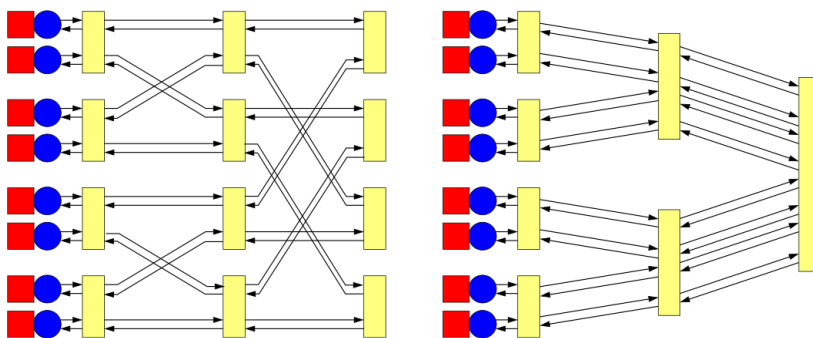
Obousměrný motýlek:

- obyčejný: kreslím na výšku
- nepřímý = můžu jen zleva doprava
(rostoucí hyperkubické dimenze)

Jdu vždycky **zleva doprava** podle e-cube směřování v hyperkrychli, a pak se rozhoduju, jestli jdu **nahoru** = 0, **dolů** = 1. (Můžu jít i zprava doleva nebo zleva doleva, pro jednoduchost ale takhle)

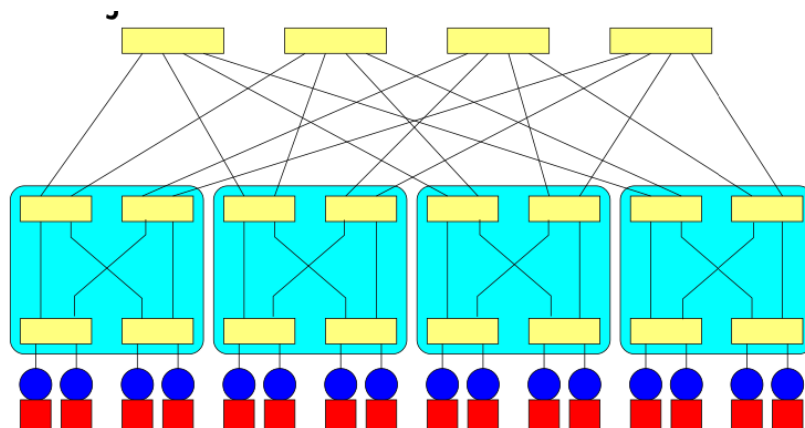


Toto směřování v podstatě odpovídá **tlustému stromu** (fat tree).



Vlevo motýlek, vpravo tlustý strom.

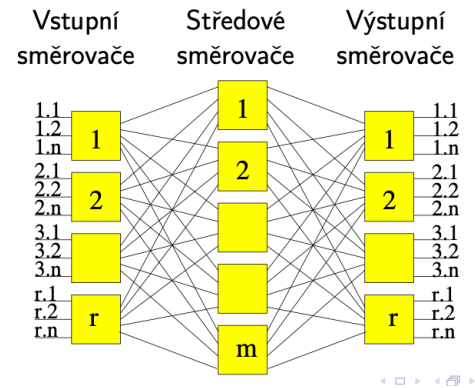
Fat tree: odpovídá hierarchii počítačů, routerů, switchů – odolný vůči výpadkům a deadlocku
- počet linek vedoucích nahoru ke kořenu se rovná součtu počtu linek od potomků



30. Closova topologie: definice, varianty, vlastnosti

3stupňová Closova topologie: $CL(m, n, r)$

- třístupňová topologie (*skládá se ze tří úrovní*), implementuje nepřímou propojovací síť $N \times N$ ($N = r \cdot n$)
- 1. stupeň: r **vstupních** směrovačů (*vstup – n , výstup – m*)
- 2. stupeň: m **středových** směrovačů (*vstup – r , výstup – r*)
- 3. stupeň: r **výstupních** směrovačů (*vstup – m , výstup – n*)

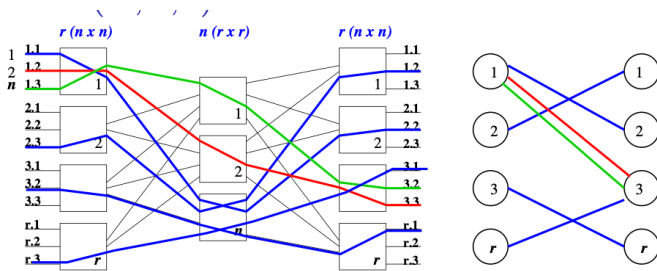
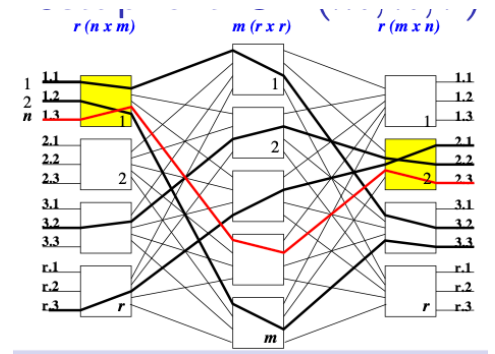


Využití: propojení serverů/racků optickými linkami s páteří datového centra

Neblokující 3stupňová $CL(m, n, r)$:

- pokud $m \geq 2n - 1$, pak je $CL(m, n, r)$ **striktně neblokující**
- lze **bezkolizně** propojit jakýkoliv volný vstupní port s jakýmkoliv volným výstupním portem

(Důkaz: již použité spoje můžou zabrat **nejvýše** $2n - 2$ středových přepínačů \Rightarrow vždycky zůstane **min. jeden** volný)



Přestavitelná $CL(m, n, r)$:

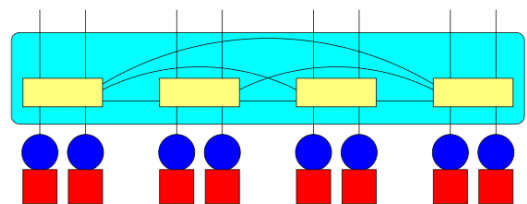
- topologie je přestavitelná, pokud $m \geq n$

(Důkaz: můžu mít plnou permutaci rn vstupů na rn výstupů – odpovídá problému barvení hran v n -árním bipartitním grafu – Hallova věta z AG2)

Tohle v té otázce asi už nebude, ale co by kdyby...

Topologie Dragonfly $DF(a, p, h)$:

- a = počet směrovačů propojených vnitřní sítí
- směrovač má p portů ke koncovým počítačům
- směrovač má h portů pro globální optické linky pro propojení s dalšími skupinami



Vlastnosti Dragonfly: každá skupina se chová jako virtuální směrovač s $a \cdot h$ porty

Směrování v Dragonfly: pokud je vnitřní síť v každé skupině úplný graf, pak mezi libovolnými směrovači existuje **min. přímá** cesta nejvýše délky 3 (1 lokální, 1 globální, 1 lokální linka)

Pokud je obsazena – využije se **nepřímá** cesta zřetěžením 2 přímých cest s náhodným mezisměrovačem = **Valiantovo** směrování.

31. Statické vnoření: definice, měřítka kvality vnoření

Statické vnoření: snažíme se vložit zdrojový graf (*procesy*) do hostitelské sítě (*procesory*)

- hledáme dvojici zobrazení $\phi: V(G) \rightarrow V(H)$ a $\xi: E(G) \rightarrow P(H)$
- vrcholy se zobrazí na vrcholy, hrany se zobrazí na cesty

Měřítka kvality vnoření:

- maximální zatížení hostitelského uzlu (*kolik procesů H v nejhorším případě poběží, **load***)
- expanze vnoření $vexp(\phi, \xi) = \frac{|V(H)|}{|V(G)|}$ (*poměr procesorů G a procesů H*)
- **dilatace** zdrojových hran (*kteřá hrana z $E(G)$ bude prodloužena na **nejdelší** cestu $P(H)$*)
- maximální zahlcení hostitelské hrany
 - a) uzlové *ncng*: počet cest začínajících, končících nebo procházejících daným uzlem grafu H
 - b) linkové *encg*: počet zdrojových cest procházejících přes cílovou hranu

Kvaziizometrické vnoření: měřítka vnoření jsou konstantní

Spodní mez na dilataci:

- pokud jsou procesy namapovány na procesory 1:1 ($|V(G)| = |V(H)|$)
 - $dil(\phi, \xi) \geq \lceil diam(H) / diam(G) \rceil$
- (Řešení bude nejlepší, když na nejdelší cestu přijdou nejvzdálenější vrcholy.)

Souvislost s MPI:

- mezi procesy vede hrana, když si pošlou právě 1 zprávu
- mapují procesy (1-D mřížka) na fyzickou topologii počítače
- předpokládám úplný graf (*ale fyzická topologie není úplná = bude zpoždění*)
- efektivnost záleží na virtuální topologii (*reálné komunikaci procesů*)
 - a) kartézská (*ortogonální graf*) – `MPI_Cart_create`
 - b) obecná (*počet uzlů, stupně, linearizovaný seznam sousedů*) – `MPI_Graph_create`
 - c) distribuovaná (*seznam sousedů*) – `MPI_Dist_Graph_create`

Problém vnoření je obecně NP-úplný.

32. Quaziizometrické topologie: mřížky – toroidy, obyčejný – uzavřený motýlek

Kvaziizometrické topologie:

- grafy G a H jsou Kvaziizometrické, pokud existuje vnoření z G do H a z H do G s konstantními hodnotami měřitek vnoření (*nezávisí na velikosti n*)


G dokáže simulovat výpočet z H se **zpomalením** h , pokud každý krok výpočtu na G lze simulovat v $O(h)$ krocích na H

G a H jsou **výpočetně ekvivalentní**, pokud G dokáže simulovat H s konstantním zpomalením a naopak

Kvaziizometrické topologie jsou **výpočetně ekvivalentní** (*naopak už to nemusí platit*).

Mřížka a toroid odpovídajících dimenzí jsou kvaziizometrické:

- mřížka je podmnožinou toroidu \Rightarrow toroid simuluje mřížku bez zpomalení
- ex. vnoření toroidu do mřížky s maximálním zatížením 1 a dilatací + linkovým zahlcením 2
 1. dekomponujeme mřížku / toroidy na jednotlivé mřížky / toroidy v jedné dimenzi
 2. vnoříme podtoroidy do podmřížek se zatížením 1 a dilatací + linkovým zahlcením 2



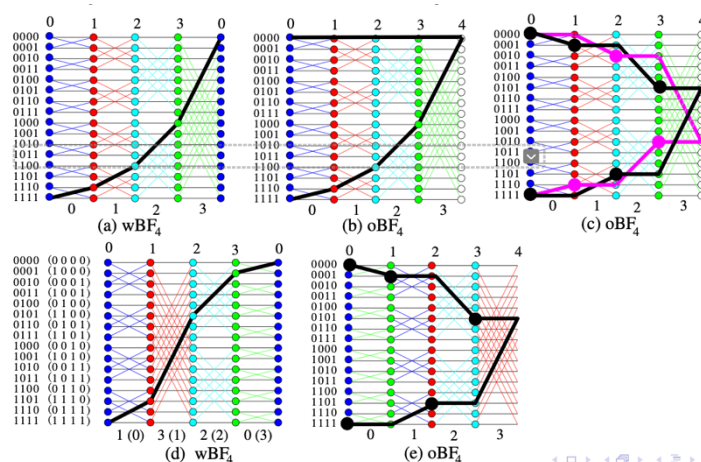
$$(a) K(5) \xrightarrow{\text{emb}} M(5).$$

3. použijeme kartézský součin na těchto podmřížkách

Obyčejný a zabalený motýlek jsou kvaziizometrické:

- obyčejný motýlek můžeme vnořit do zabaleného s max. zatížením 2 a dilatací 1 (*sloučím koncový uzly jednotlivých řad otevřeného motýlku a získám kružnice*)
- zabalený motýlek můžu vnořit do otevřeného s max. zatížením 1 a dilatací ≤ 3

(logika: vezmu si nějakou cestu na zabaleném motýlku, chci ji namapovat na obyčejný; někdy to jde triviálně s dilatací 1, někdy musím provést vhodně permutaci vrcholů tak, aby se mi to natáhlo maximálně na 3)



Důsledek: wBF_n má $n!$ automorfismů daných permutacemi bitů v řádkových adresách
 oBF_n má $n!$ automorfismů daných permutacemi bitů v adresách řádků

33. Vnoření hyperkrychle do nízkorozměrných mřížek

Vnoření hyperkrychle:

- uvažujeme pouze vnoření $Q_{2^k} \rightarrow M(2^k, 2^k)$ s max. zátěží 1, kde $k \geq 2$ je přirozené číslo
- dolní mez na dilataci takového vnoření je podíl průměrů: $\frac{2^{k-1}}{k}$

Realizace mapování Q_n do 2-D mřížky: mapa logické funkce

- Svobodova mapa (*lexikografická indexace*)
- Karnaughova mapa (*Grayova indexace*)

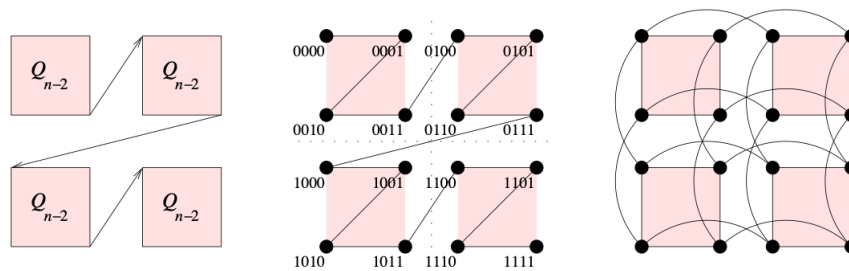
Mapují vlastně $2k$ -bitové adresy hyperkubických uzlů na jednotlivé uzly v mřížce

- lexikografické mapování po **řádcích**: $\phi(b_{2k-1}b_{2k-2} \dots b_0) = [b_{2k-1} \dots b_k, b_{k-1} \dots b_0]$
- lexikogr. mapování po **sloupcích**: $\phi(b_{2k-1}b_{2k-2} \dots b_0) = [b_{k-1} \dots b_0, b_{2k-1} \dots b_k]$
- **Mortonova křivka**: $\phi(b_{2k-1}b_{2k-2} \dots b_0) = [b_{2k-1}b_{2k-3} \dots b_1, b_{2k-2}b_{2k-4} \dots b_0]$

Po řádcích = horní půlku bitů dám do jedné dimenze, dolní půlku dám do druhé.

Po sloupcích = obráceně, „transpozice“ toho, co jsem udělal po řádcích.

Mortonova: střídavě rekurzivně mapuji ve směru osy x a y – vytvoří fraktální strukturu.



Indukční krok, vnoření vrcholů a vnoření hran

Platí, že vzdálenost 2 uzlů lišících se v bitu i , je $2^{\lfloor \frac{i}{2} \rfloor} \Rightarrow$ **dilatace** vnoření Mortonovou křivkou je 2^{k-1} .

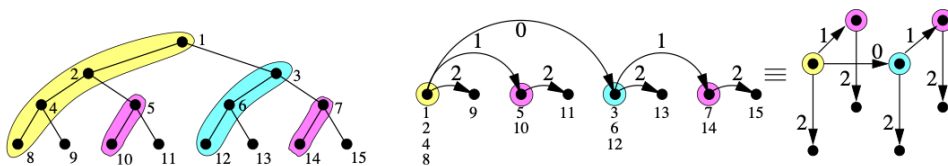
34. Implementace paralelního Rozděl-a-panuj v hyperkrychlích a mřížkách

Implementace rozděl a panuj v hyperkrychlích:

- nativní topologie pro realizaci výpočtu **rozděl-a-polovinu-si-nech**
- logika výpočtu odpovídá **binominální kostře** hyperkrychle

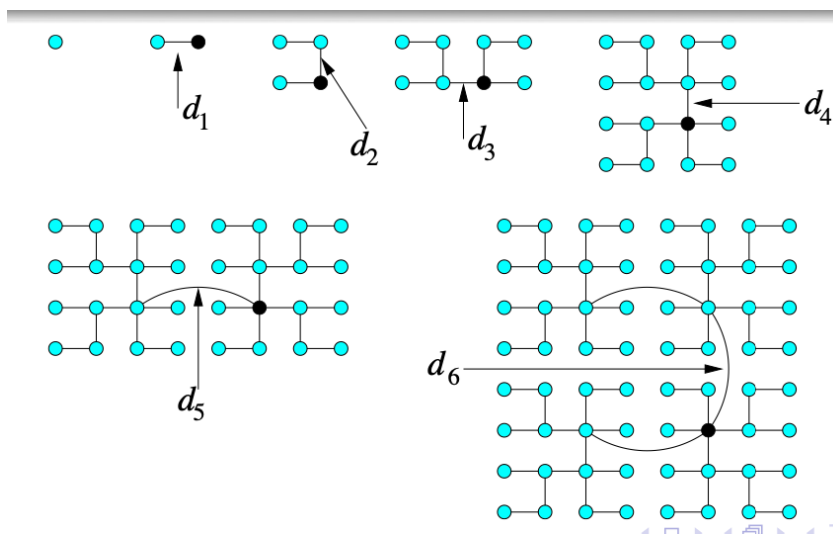
Algoritmus (hyperkrychle):

1. kořen rozdělí problém do 2 polovin, 1/2 předá sousedu v dimenzi 0, 1/2 si nechá
2. oba dva udělají totéž s použitím dimenze 1, pak dimenze 2, ...
3. všechny uzly Q_n provedou řešení listových podproblémů



Algoritmus (mřížka):

- počítáme s mřížkou $M(2^{\lfloor \frac{n}{2} \rfloor}, 2^{\lfloor \frac{n}{2} \rfloor})$ realizující rekurzivní n -úrovňový výpočet
 - každý uzel mřížky realizuje jeden list stromu s dilatací 1 pro $n \leq 4$ a
- $$d_n = 2(2^k + 2^{k-2} + 2^{k-4} + \dots + 2^{k \bmod 2} + 1, \text{ kde } k = \left\lfloor \frac{n-5}{2} \right\rfloor \text{ pro } n \geq 5$$



Tato mřížka **simuluje** Q_n , provádíme tedy vnoření binominální **kostry** Q_n do 2-D mřížky. Vnoření systematicky **střídá** horizontální a vertikální dimenzi mřížky s použitím zrcadlové symetrie.

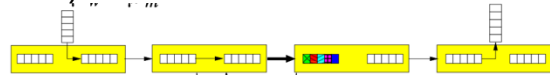
(Při provádění algoritmu tedy uzel půlku dat pošle do druhé dimenze, ten to zase pošle takhle dál – je to 2D mřížka, tedy opravdu to, co vidíme na obrázku)

35. Technologie přepínání v paralelních počítačích ulož-pošli-dál a červí: popis, komunikační latence

Metriky:

- **šířka kanálu** w : počet bajtů, které lze najednou převést mezi 2 sousedními směrovači
- **zpoždění kanálu** t_m : zpoždění mezi sousedními směrovači
- **startovní latence** t_s : SW a HW zpoždění v zdrojovém a cílovém uzlu, příprava síťové komunikace
- **směrovací latence** t_r : čas pro směrovací rozhodnutí během budování trasy
- **přepínací latence** t_w : čas přenosu v přepínači ze vstupních na výstupní kanály
- **komunikační latence**: síťová (*směrovací, přepínací*) latence + startovní latence

Store-and-forward (ulož-pošli-dál):



- zprávy rozděleny do packetů **pevné** délky, packety rozděleny do **flitů** (první hlavičkový flit)
- **individuální** směrování ze zdroje do cíle, přepínání packetů
- vstupní a výstupní fronty pro **celé** packety, předpokládáme dostatečnou kapacitu
- směrovač obdrží **celý** packet, rozhodne, kam ho přepne, a přepośle ho dál
- **hop**: zkopírování celého packetu z výstupní fronty 1 směrovače do vstupní fronty dalšího

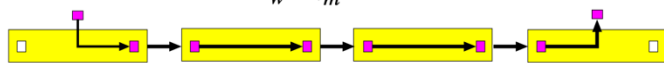
Latence přenosu packetu délky μ na vzdálenost δ (SAF): $t_{SF}(\mu, \delta) = t_s + \delta(t_r + (t_w + t_m)\mu)$

- startovní latence, pak v každém přepínači musím rozhodnout a překopírovat na další

Využití:

- vhodné pro **krátké a časté** zprávy (*z celé trasy obsažen nejvýše 1 kanál*)
- síťová latence úměrná součinu velikosti packetu a délky trasy
- požadavek na **malý průměr** propojovací sítě a směrování po **nejkratších cestách**

Wormhole (červí přepínání):



- zprávy rozděleny do packetů, ty do **flitů**
- směrovače nemají fronty pro celé packety, ale pouze pro **1 flit**
- po příchodu hlavičkového flitu se nečeká na uložení packetu, ale okamžitě se **prořízne** dál
- bezkolizní packet = řetězec za sebou jdoucích flitů, jiné požadavky **zablokovány**

Latence: $t_{WH}(\mu, \delta) = t_s + \delta(t_r + t_w + t_m) + \mu \max(t_w, t_m)$

- startovní latence, pak se prořízne hlavička na celou trasu, zbytek packetu se už nerozhoduje

Využití:

- nezávisí na vzdálenosti = vhodné pro **dlouhé** zprávy
- je citlivé na **zablokování** – zprávy nesmí být příliš časté
- není příliš požadavků = propojovací síť může být malá, levná a rychlá

36. OAB ve všeportových a 1-portových SF sítích: spodní meze složitosti, algoritmy, jejich složitosti

Parametry modelů KKO:

- jednoportový (MPI_THREAD_SERIALIZED), všeportový (MPI_THREAD_MULTIPLE)
(Na daném uzlu běží pouze jedno vlákno, nebo více vláken?)
- **směrovost kanálů**: plně-duplexní, poloduplexní
- **technika přepínání**: uloži-a-pošli-dál nebo červí
- manipulace se zprávami: **nekombinující** (každá zpráva putuje sítí zvlášť),
kombinující (zprávy jdoucí stejným směrem se slučují a v cíli se rozdělí)

Počet paralelních kroků: spodní (ró) mez $\rho_{XXX}(G)$, horní mez $r_{XXX}(G)$

- SF síť: 1 krok = množina **bezkolizních** hopů (zkopírování mezi frontami směrovačů)
- WH síť: 1 krok = množina současně použitých linkově disjunktních cest

Komunikační latence: spodní (τ) mez $\tau_{XXX}(G)$, horní mez $t_{XXX}(G)$

Komunikační práce: spodní (η) mez $\eta_{XXX}(G)$, horní mez $h_{XXX}(G)$

- celkový počet **hopů** (SF) nebo **paketo-hran** (WH)

Efektivní algoritmy:

- využívají maximální kapacitu sítě v co nejvíce krocích algoritmu
- eliminují **redundantní** komunikaci
 - NODUP** (žádný uzel nedostane tutéž informaci 2x)
 - NOHO** (žádný uzel neobdrží zpět zprávu, kterou již sám odeslal)

Optimální algoritmy: horní a spodní meze jsou asymptoticky stejné

One to All Broadcast ve SF sítích, k -portový model:

- komunikační práce $\eta_{OAB,k}^{SF}(G, s) = |V(G)| - 1$ (každý cílový uzel musí 1 hopem přijmout)
- počet paralelních kroků $\rho_{OAB,k}^{SF}(G, s) = \max(\text{exc}(s, G), \log_{k+1} |V(G)|)$
(počet informovaných uzlů se v každém kroku může nejvýše k -násobit – proto logaritmus, informace se ale předává pouze sousedům, proto nemůže překročit excentricitu)
- spodní mez na součet maxim délek paralelních cest přes všechny kroky algoritmu:
 $\gamma_{OAB,k}^{SF}(G, s) = \text{exc}(s, G)$ (v každém kroku nejdelší cesta – musí překročit excentricitu)
- spodní mez komunikační latence: $\tau_{OAB,k}^{SF}(G, s) = \rho_{OAB,k}^{SF}(G, s)(t_s + \mu t_m) + \gamma_{OAB,k}^{SF}(G, s)t_d$
(v každém kroku posílám packet o velikosti μ po linkách s rychlostí přenosu t_m)

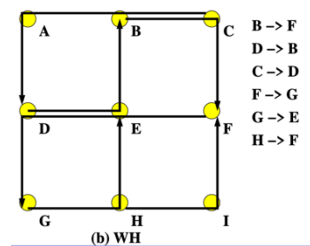
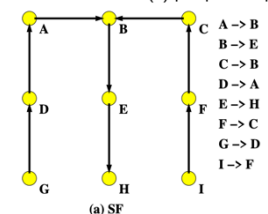
V uzlově symetrických sítích: $\text{exc}(s, G) = \text{diam}(G)$ – souřadnice neovlivní složitost.

Optimální OAB ve všeportových SF sítích:

- záplavový krokově-optimální OAB algoritmus
- zdroj pošle packet všem sousedům, ty buď přijmou a pošlou dále, nebo ho ignorují
- pro zajištění NODUP+NOHO – nutné použít kostru nejkratších cest = **řízená záplava**

Optimální OAB v jednoportových SF sítích:

- v nejlepším případě se počet informovaných uzlů zdvojnásobí, obecně NP-úplný problém
- **rekurzivní zdvojování**: rozdělím graf do dvou podgrafů, a paralelně posílám po těchto dvou
- souvisí s binárním D&C na hyperkrychli Q_n (vytvářím binominální kostru), lze i na mřížce



37. OAB v 1-portových WH sítích: spodní meze složitosti, algoritmy, jejich složitosti

One to All Broadcast ve WH sítích, k portový model:

- WH: vzdálenost nemá vliv, klíčový je pouze počet portů
- komunikační práce $\eta_{OAB,k}^{WH}(G, s) = |V(G)| - 1$ (každý cílový uzel musí 1 hopem přijmout)
- počet paralelních kroků $\rho_{OAB,k}^{WH}(G, s) = \log_{k+1} |V(G)|$
(počet informovaných uzlů se v každém kroku může nejvýše k -násobit – proto logaritmus)
- spodní mez na součet maxim délek paralelních cest přes všechny kroky algoritmu:
 $\gamma_{OAB,k}^{WH}(G, s) = exc(s, G)$ (v každém kroku nejdelší cesta – musí překročit excentricitu)
- spodní mez komunikační latence: $\tau_{OAB,k}^{WH}(G, s) = \rho_{OAB,k}^{WH}(G, s)(t_s + \mu t_m) + \gamma_{OAB,k}^{WH}(G, s)t_d$
(v každém kroku posílám packet o velikosti μ po linkách s rychlostí přenosu t_m)

Optimální algoritmus v 1D toroidech a mřížkách: hyperkubické rekurzivní zdvojování

- zdroj s rozdělí toroid $K(z)$ do dvou polovin, nechá si menší (z liché) / větší část, druhou pošle
- rekurzivně se opakuje v obou dvou polovinách současně

Optimální algoritmus v více-D mřížkách:

- s pomocí kartézského součinu rozložíme do více dimenzí
- směruji postupně podle pořadí dimenzí

38. OAB ve všeporťovém WH 2-D toroidu: spodní mez složitosti, algoritmus zobecněné diagonály, jeho složitost

One to All Broadcast ve WH sítích, k portový model:

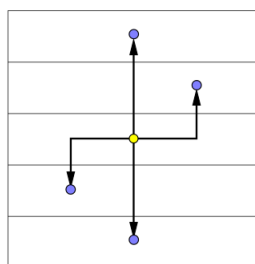
- WH: vzdálenost nemá vliv, klíčový je pouze počet portů
- komunikační práce $\eta_{OAB,k}^{WH}(G, s) = |V(G)| - 1$ (každý cílový uzel musí 1 hopem přijmout)
- počet paralelních kroků $\rho_{OAB,k}^{WH}(G, s) = \log_{k+1} |V(G)|$
(počet informovaných uzlů se v každém kroku může nejvýše k -násobit – proto logaritmus)
- spodní mez na součet maximálních délek paralelních cest přes všechny kroky algoritmu:
 $\gamma_{OAB,k}^{WH}(G, s) = exc(s, G)$ (v každém kroku nejdelší cesta – musí překročit excentricitu)
- spodní mez komunikační latence: $\tau_{OAB,k}^{WH}(G, s) = \rho_{OAB,k}^{WH}(G, s)(t_s + \mu t_m) + \gamma_{OAB,k}^{WH}(G, s)t_d$
(v každém kroku posílám packet o velikosti μ po linkách s rychlostí přenosu t_m)

Počet paralelních kroků zde odpovídá $\lceil \log_{2n+1} N \rceil$ pro n -rozměrnou všeporťovou mřížku

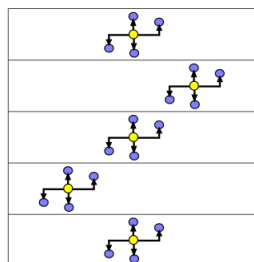
- aby byla spodní mez dosažena, musí v každém kroku uzel najít $2n$ neinformovaných uzlů
- doručit jim packet po hranově disjunktích cestách se všemi aktuálně existujícími cestami
- díky uzlové symetrii řešitelné pro **2-portový 1-D toroid** a **4-portový 2-D toroid**

Algoritmus zobecněné diagonály (pro OAB v 4-portovém 2-D toroidu):

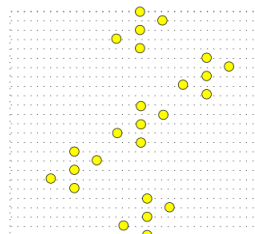
- 1) zdrojový uzel doručí 1 packet do každého řádku
 - a) rozdělíme toroid do 5 horizontálních pásů přibližně stejné šířky
 - b) ze zdrojového pásu pošleme do všech ostatních 4 pásů hranově disjunktími cestami XY směřováním => rekurzivně, dokud není doručeno do každého řádku
- 2) seřazení packetů na hlavní diagonále paralelně ve všech řádcích
- 3) diagonální uzly informují všechny zbývající uzly
 - a) rozdělíme toroid do 5 diagonálních pásů přibližně stejné šířky
 - b) z hlavní diagonály pošleme do všech ostatních 4 diagonál hr. disj. cestami => rekurzivně



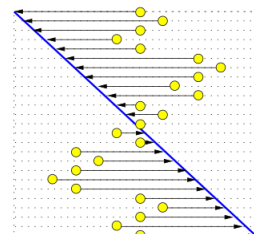
Phase 1 – Round 1



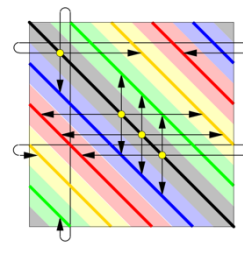
Phase 1 – Round 2



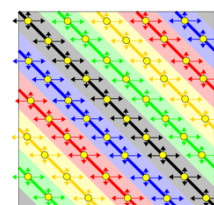
The result of Phase 1



Phase 2



Fáze 3 – krok 1



Fáze 3 – krok 2

39. Multicast v 1-portové WH 2-D mřížce: spodní mez složitosti, optimální algoritmus, jeho složitost

Multicast: jeden uzel posílá stejnou zprávu zadané podmnožině M uzlů propojovací sítě G

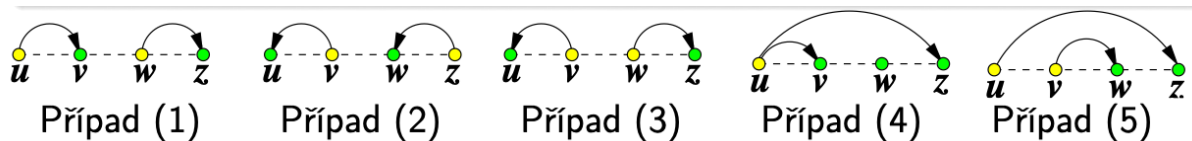
- obtížnější než WH OAB, použití OAB algoritmu neefektivní, pokud M výrazně menší
- spodní meze analogické spodním mezím OAB vztaheným na M

- komunikační práce $\eta_{MC,k}^{WH}(G, s) = |M| - 1$ (každý cílový uzel musí 1 hopem přijmout)
- počet paralelních kroků $\rho_{MC,k}^{WH}(G, s) = \log_{k+1} |M|$
(počet informovaných uzlů se v každém kroku může nejvýše k -násobit – proto logaritmus)
- spodní mez na součet maximálních délek paralelních cest přes všechny kroky algoritmu:
 $\gamma_{MC,k}^{WH}(G, s) = exc(s, G, M)$ (v každém kroku nejdelší cesta – musí překročit excentricitu)
- spodní mez komunikační latence: $\tau_{MC,k}^{WH}(G, s) = \rho_{MC,k}^{WH}(G, s)(t_s + \mu t_m) + \gamma_{MC,k}^{WH}(G, s)t_d$
(v každém kroku posílám packet o velikosti μ po linkách s rychlostí přenosu t_m)

Multicast v jednoportové WH 2-D mřížce:

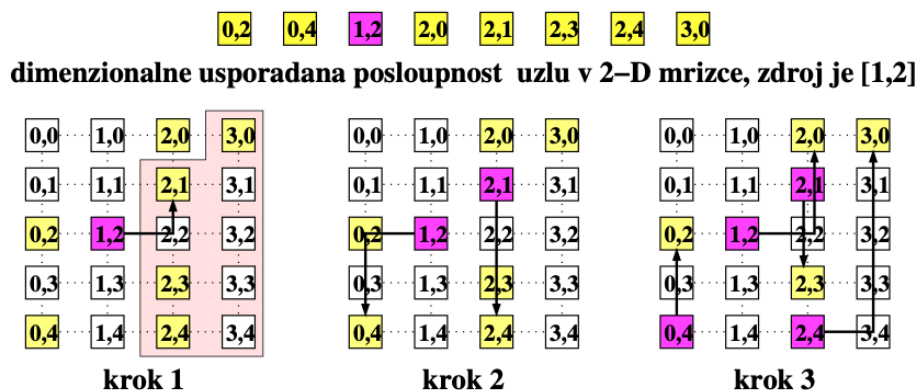
- uvažujeme XY směřování (nejprve pohyb v horizontálním směru X , pak vertikálním Y)
- **dimenzionálně uspořádaná posloupnost:** seřazené lexikograficky dle souřadnic
Příklad takové posloupnosti: $[0,3] < [1,1] < [1,3] < [2,4] < [3,0]$

Pokud máme čtveřici uzlů $u < v < w < z$ a jsou dimenzionálně uspořádané, pak všechny cesty jsou buď hranově disjunktní, nebo zadefinujeme přednost ($u \rightarrow z$ před $u \rightarrow v$ nebo $v \rightarrow w$):



Optimální algoritmus: rekursivní zdvojování nad virtuální 1-D mřížkou

- 1) rozdělíme dimenzionálně uspořádanou posloupnost na levou a pravou polovinu
- 2) zdroj v levé polovině => pošle packet prvnímu uzlu v pravé, jinak poslednímu uzlu v levé
- 3) rekursivně opakujeme na obě poloviny



(Ty poloviny se určují podle toho, kterým uzlům to chceme poslat, takže tady v levé polovině jsou $[0,2]$ $[0,4]$ $[1,2]$ $[2,0]$ a v pravé polovině $[2,1]$ $[2,3]$ $[2,4]$ $[3,0]$)

40. Kombinující OAS: spodní meze, algoritmy, jejich složitosti

One to All Scatter:

- zdrojový uzel posílá **individuální** packet (*stejně velikosti*) každému uzlu
 - zdroj tedy vysílá $N - 1$ různých packetů velikosti μ a každý uzel získá 1 takový packet
 - otočíme-li orientaci, získáváme All to One Gather [AOG] (= *obdobná složitost*)
- AOG: cíl má obdržet $N - 1$ různých packetů, OAS: zdroj má rozeslat $N - 1$ různých packetů
AAG / AAB: každý má obdržet $N - 1$ různých packetů

Kombinující OAS, k-portový model:

- počet paralelních kroků $\rho_{OAS,k}(G, s) = \rho_{OAB,k}(G, s)$ (*dvojnásobí se / excentricita u SF*)
- spodní mez na součet maxim délek paralelních cest přes všechny kroky algoritmu:
 $\gamma_{OAS,k}(G, s) = exc(s, G)$ (*v každém kroku nejdelší cesta – musí překročit excentricitu*)
- spodní mez kom. latence: $\tau_{OAS,k}(G, s) = \rho_{OAS,k}(G, s)t_s + \gamma_{OAS,k}(G, s)t_d + \left\lceil \frac{|V(G)|-1}{k} \right\rceil \mu t_m$
(v každém kroku posílám $|V(G)| - 1$ packetů o velikosti μ po k linkách s rychlostí přenosu t_m)

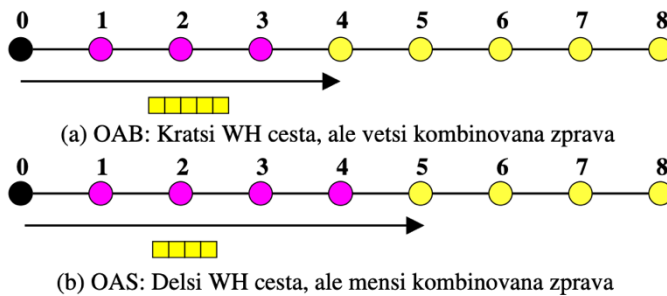
Platí: kombinující OAS je stejný jako OAB, akorát velikost zprávy se postupně zmenšuje
(když uzel dostane $N - 1$ zpráv, dál posílá už jen $N - 2$ [tu svoji už obdržel])

Optimální algoritmus (1-portová hyperkrychle):

- komunikační struktura: SF binominální kostra (*stejně jako u OAB*)
- velikost zprávy klesá v každém kroku na polovinu
- $t_{OAS}(Q_n, \mu) = t_s n + \mu t_m (2^n - 1)$ (*v podstatě odpovídá obecnému vzorci*)

Optimální algoritmus (WH, mřížky a toroidy): opět rekurzivní zdvojování

- pro větší zprávy je ale efektivnější volit delší cesty s menšími zprávami



- $t_{OAS}(M(z), \mu, 0) = t_s \lceil \log(z) \rceil + (t_d + \mu t_m)(z - 1)$ (*opět odpovídá obecnému vzorci*)

41. Nekombinující AAB/AAG: spodní meze, časově hranově disjunktní stromy a hranově-disjunktní hamiltonovské kružnice, algoritmy, jejich složitosti

All to all broadcast / gather: každý uzel posílá každému uzlu stejnou zprávu

Nekombinující AAB/AAG, k-portový model:

- počet paralelních kroků $\rho_{AAB,k}(G) = \left\lceil \frac{|V(G)|-1}{k} \right\rceil$
(každý uzel musí přijmout $|V(G)| - 1$ packetů a v jednom kroku jich může přijmout nejvýše k)
- spodní mez komunikační latence: $\tau_{AAB,k}(G) = \rho_{AAB,k}(G)(t_s + \mu t_m)$
(jeden krok nemůže být menší než přijetí 1 packetu od souseda)

Algoritmus pro nekombinující SF AAB/AAG:

- vyžaduje všeportovou plně-duplexní síť, vychází z OAB stromu (kostry)
- všechny uzly začnou OAB ve stejném okamžiku a postupují synchronně toutéž rychlostí

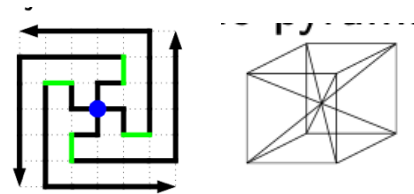
Časově-hranově-disjunktní stromy (TADT) – vyžaduje SF a uzlovou symetrii:

- orientovaná hrana je úrovně i , pokud přes ni projde packet v kroku i
- výška stromu $h(B(u))$ je číslo nejvyšší úrovně hrany v $B(u)$
- dva stromy $B(u)$ a $B(v)$ jsou **časově-hranově-disjunktní stromy**, pokud pro každé i jsou množiny jejich hran na úrovni i disjunktní

(Lidsky řečeno: pokud v daném okamžiku prochází přes danou hranu – spojení mezi dvěma uzly / směrovači jen jeden packet, pak jsou jejich stromy disjunktní, a přenos je **bezkolizní**).

SF Algoritmus (TADT) pro 2-D a 3-D toroidy:

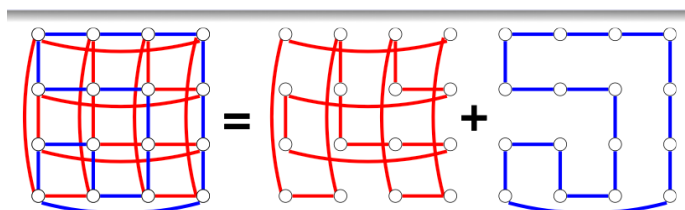
- pro liché z u toroidů $K(z,z)$ nebo $K(z,z,z)$ je triviálním řešením rotace 2-D / 3-D hada
- v obecném případě je toto řešení komplikovanější



(Lidsky: $B(*)$ mi rozdělí toroid na 4/6 disjunktní podstromy. V jednom kroku pošlu zprávy tak, v druhém orotuji a pošlu znovu, ... až do posledního kroku, pak využiju uzlovou symetrii a jedu znovu).

SF algoritmus pro 2-D toroidy: hranově disjunktní hamiltonovské kružnice

- 1) zkonstruujeme 2 hranově disjunktní hamiltonovské kružnice H_1 a H_2
- 2) každý uzel rozpůlí packet p_i na 2 **polopackety**, ty pošle po H_1 a H_2
- 3) každý uzel přijme 4 polopackety, přepošle je dále, a pokud již obdržel obě dvě části nějakého packetu j , složí je zpět do původního packetu



Každý polopacket cestuje jen do **půlky** své kružnice a tedy $t_{AAB}(K(z_1, z_2)) = \frac{z_1 z_2}{2} (t_s + \frac{\mu}{2} t_m)$

42. AAS: spodní meze komunikační latence

All to All Scatter: úplná výměna, osobní komunikace všichni-všem

- na počátku každý uzel vlastní $N - 1$ packetů velikosti μ (*pro každý uzel jeden*)
- na konci má každý uzel N jiných packetů (*jeden od každého jiného*)
- dochází k výměně $N(N - 1)$ packetů
- například transpozice matice mapované po řádcích na N -procesorový počítač

Spodní mez AAS (síťová propustnost): $\tau_{AAS}(G, \mu) = \frac{1}{2|E(G)|} (\sum_{u \neq v} dist_G(u, v)) \mu t_m$

- operace požaduje přenést packet mezi každou uspořádanou dvojicí různých uzlů u a v
- to vyžaduje součet vzdáleností uzlů pro disjunktí uzly, kapacita je ale pouze $2|E(G)|$

Spodní mez AAS (bisekční šířka): $\tau_{AAS}(G, \mu) = \frac{\lceil N/2 \rceil \lfloor N/2 \rfloor \mu t_m}{bw_e(G)}$

- vrcholy můžu rozdělit na dvě poloviny, během AAS každý uzel z první poloviny pošle zprávu každému uzlu z druhé poloviny a naopak
- počet hran mezi polovinami je v nejhorším případě $bw_e(G)$
- komunikace pak musí projít přes hrany bisekčního řezu velikosti $bw_e(G)$

Bonus: KKO v MPI:

OAB: MPI_Bcast(void *data, int cnt, MPI_Datatype, int root, MPI_Comm)
AOG: MPI_Gather(const void * send, int sndcnt, MPI_Datatype sndtype,
void * rcv, int rcv_cnt, MPI_Datatype rcv_type, int root, MPI_Comm)

root je proces, který posílá/přijímá data, AOG **sbírá** data od všech procesů včetně sebe sama
By default se sbírá stejný počet dat od každého procesu, pokud ne, varianta MPI_Gatherv

AAG/AAB: MPI_Allgather (*syntaxe podobná MPI_Gather, ale data sbírají všichni*)

OAS: MPI_Scatter(const void * send, int sndcnt, MPI_Datatype sndtype,
void * rcv, int rcv_cnt, MPI_Datatype rcv_type, int root, MPI_Comm)

OAS = Obrácená operace k MPI_Gather, root **distribuuje** data procesům včetně sebe sama

AAS: MPI_Scatter(const void * send, int sndcnt, MPI_Datatype sndtype,
void * rcv, int rcv_cnt, MPI_Datatype rcv_type, MPI_Comm)

U AOG, OAS a AAS lze použít MPI_IN_PLACE, pak se využije receive buffer i jako send buffer.

43. Paralelní redukce: implementace na PRAM a různých topologiích a jejich složitost, škálovatelnost, MPI funkce

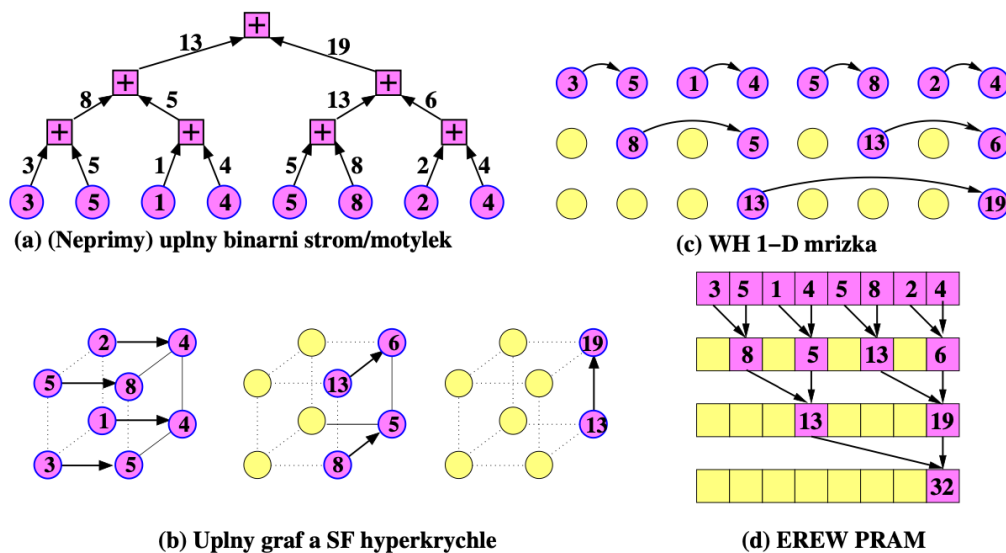
Paralelní redukce:

- vstupem je pole n prvků z množiny D , množina p procesorů a **asociativní** operace \oplus
- výstupem je skalár $S = X[0] \oplus X[1] \oplus \dots \oplus X[n-1]$
- odpovídá **All-to-One reduction**

Složitost a škálovatelnost:

- paralelní čas: $O(n/p + \log(p))$, dolní mez: $L(n, n) = \Omega(\log n)$
- dobrá škálovatelnost: $\psi_1(p) = p \log(p)$, $\psi_2(n) = n/\log(n)$
- hyperkubický algoritmus = **optimální** na hyperkubických sítích

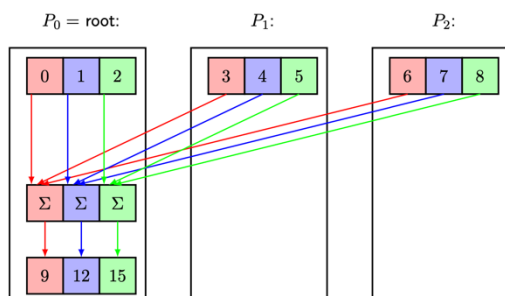
Implementace:



Na binárním stromu nebo motýlkovi postupně od listů redukuji směrem nahoru.
 V hyperkrychli dělám v podstatě opak D&C – posílám postupně směrem na jeden uzel.
 Ve WH mřížce posílám ob uzly, podobně v EREW PRAM.

MPI funkce: `MPI_Reduce(const void *send, void *recv, int count, MPI_Datatype, MPI_OP, int root, MPI_Comm)`

- výsledek bude zapsán do parametru `recv` u procesu `root`, opět možnost `MPI_IN_PLACE`
- MPI definuje vhodné kombinace operace `MPI_OP` a typu operandů `MPI_Datatype`



Alternativně: `MPI_Allreduce`

- neobsahuje parametr `root`, všechny procesy obdrží výsledek
- All-to-all reduction (**AAR**)

Nebo: `MPI_Reduce_scatter_block`

- každý proces obdrží výsledek redukce i-tých bloků

44. Paralelní prefixový součet: definice, implementace na PRAM, APRAM a různých topologiích a jejich složitosti, škálovatelnost, MPI funkce

Paralelní prefixový součet:

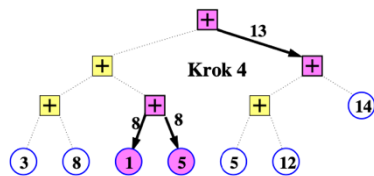
- zobecnění redukce, opět máme pole n prvků z množiny D , asoc+kom. operace \oplus
- výstupem je pole Y prefixových součtů: $Y[i] = X[0] \oplus X[1] \oplus \dots \oplus X[i]$
- předpokládáme, že na jeden procesor připadá jedno číslo

Implementace na PRAM / APRAM:

- v 1. kroku přičtu do každé buňky hodnotu o 1 vedle
- v 2. kroku o 2 nalevo, ve 3. o 4 nalevo, 4. o 8 nalevo...

Implementace na nepřímém stromu / obousm. motýlku:

- vstupní data v listech, vnitřní uzly počítají
- výsledek se pak vrátí do listů
- **algoritmus:** když dostanu hodnotu, pošlu ji do pravého podstromu a nahoru – rekurzivně na každém uzlu



Na přímém stromu: použiju POSTORDER, posílám hodnoty zleva doprava.

Složitost: $2h(T)$ kroků, kde $h(T)$ je výška stromu T .
(Cesta nahoru a dolů)

Důsledek: na jakémkoliv souvislém grafu, u kterého dokážu zkonstruovat kostru, jsem schopný řešit PPS v $O(\text{diam}(G))$ krocích.

Implementace na hyperkrychli:

- postupně vyměňuji hodnoty v sousedních uzlích
- držím si mezisoučet (žluté) a celkový součet (zelené)
- výsledek je uložen v mezisoučtu

Implementace na SF mřížkách:

- provedeme linearizaci, například po řádcích shora dolů
- střídají se svislé a vodorovné fáze

Implementace na WH mřížkách:

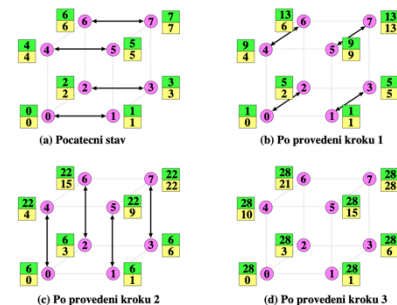
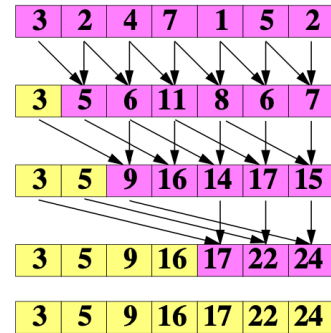
- simulujeme prefixový součet na nepřímém binárním stromu (*namapuji mřížku na strom*)

Škálovatelnost:

- můžu provést lokální předvýpočet, pak globalizaci a dopočet globálního
- složitost: $T(n, p) = O(n/p) + O(\log p)$, škálovatelnost **stejná** jako paralelní redukce (*PRAM, hyperkrychle, WH mřížky, síť s log průměrem*)

Implementace na APRAM: synchronizace fází, $O(n/p + \text{bariéra} \log^2 p)$

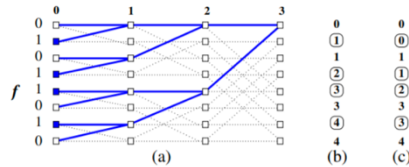
MPI: MPI_Scan(const void *send, void *recv, int count, MPI_Datatype, MPI_OP, MPI_Comm), případně MPI_Exscan pokud P_i obsahuje prefix nad daty v $P_0..P_{i-1}$



45. Aplikace paralelního prefixového součtu: zhušťovací problém, paralelní RadixSort, sčítačka s predikcí přenosu, tridiagonální systém lineárních rovnic

Zhušťovací problém:

- výpočet pořadí v distribuované podmnožině Z daného univerza
- máme např. červené a modré prvky, chceme spočítat, kolikátý červený prvek v poli to je
- řešení: prefixový součet nad binárním polem (*charakteristickým vektorem*)



(a) Počáteční hodnoty příznaků f_i a jeden zvolený nepřímý strom.

(b) Pole příznaků f_i po PPS.

(c) Konečné hodnoty indexů cílových výstupů.

Paralelní RadixSort:

- řadím podle nějaké předpony, provádím zhuštění nahoru (0) / dolů (1)
- iterativně opakuji dvě zhuštění a dva prefixové součty

Paralelní binární sčítačka s predikcí přenosu:

- mám dvě n -bitová binární čísla, chci předpočítat přenosové bity
- provádím prefixový součet nad polem bitů b_i , kde:
 $b_i = \text{generate (pokud } x_i = y_i = 1) / \text{stop (} x_i = y_i = 0) / \text{propagate (jinak)}$
- za použití speciální operace (stop / gen + cokoliv = stop / gen, p + cokoliv = cokoliv)

Tridiagonální systém lineárních rovnic:

- odpovídá procesu zrodu a zániku (*NI-VSM*)
- přepíšu si i -tou rovnici na maticové násobení, z toho dostanu rekurentní tvar
- dílčí matice H_i počítám prefixovým součinem matic ($H_i = G_i G_{i-1} \dots G_1$)

46. Segmentovaný paralelní prefixový součet: definice, implementace, aplikace pro paralelní QuickSort

| 2 1 3 5 | 2 7 3 | 9 4 5 6 | 2 8 4 3 1 |
 | 2 3 6 11 | 2 9 12 | 9 13 18 24 | 2 10 14 17 18 |

Segmentovaný paralelní prefixový součet:

- vstup: pole rozdělené do **segmentů** libovolné délky
- výstup: prefixové součty uvnitř těchto segmentů, **izolovaně**

Implementace: jako standardní prefixový součet se speciálně zadanou operací

- $(a, b) = (a, b)$ $(|a, b) = |(a, b)$ $(a, |b) = |b$ $(|a, |b) = |b$

\oplus	b	$ b$
a	$a \oplus b$	$ b$
$ a$	$ (a \oplus b)$	$ b$

Aplikace pro paralelní QuickSort:

- uvažujeme stabilní sekvenční out-of-place Quicksort, paralelizujeme segmentovaným PPS
- vstupní posloupnost rovnoměrně rozdělujeme na tři segmenty $<, =, >$
- pomocí zhuštění zhuštím čísla v segmentech (*předpočítám indexy*) a přemístím
- pivota si můžu předpočítat pomocí speciální operace $a + b = a$ v segmentovaném PPS

47. Paralelní systémy souborů a paralelní zápis velkých výstupních dat všemi procesy do společného výstupního souboru pomocí MPI I/O

Paralelní vstup a výstup:

- čtení ze souboru odpovídá příjmu zpráv, zápis do souboru posílání zpráv
- potřebujeme **paralelní souborový systém** (Lustre, GPFS)
- vyplatí se pro **velké** soubory, pro velké množství relativně malých souborů NE
- části souboru mapovány na koncová zařízení (*u Lustre Object Storage Target*)

MPI I/O:

- soubor je reprezentován jako `MPI_File`
- operace `MPI_File_open`, `MPI_File_close`, `MPI_File_seek`, `MPI_File_read`, `MPI_File_write`

Ošetření chyb v MPI-I/O:

- u komunikačních operací se váže na MPI komunikátor, viz otázka 22
- u souborových operací se váže na soubor `MPI_File`, implicitně `MPI_ERRORS_RETURN` (*možnost nastavit jinou obsluhu přes `MPI_File_set_errhandler`*)

Implementace:

- pomocí prefixového součtu `MPI_Exscan` každý procesor zjistí, kde může začít zapisovat
- pak použije `MPI_File_open`, `MPI_File_write_at` a `MPI_File_close`

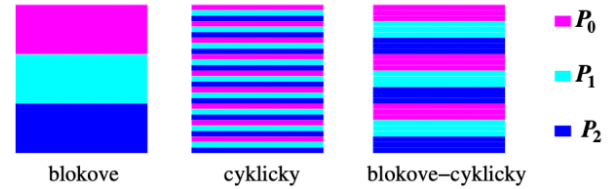
Nad MPI-I/O existují i knihovny, které umožňují pracovat se soubory pohodlněji: **HDF5**

48. Násobení husté matice vektorem při řádkovém, sloupcovém a šachovnicovém mapování

Matice: uvažují pouze čtvercové husté matice, reprezentované jako 2-D pole

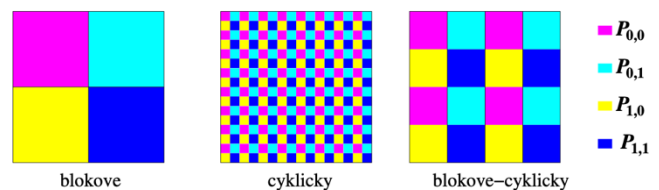
Mapování matic na procesy – proužkové:

- každý procesor má matici uloženou po proužkách, procesy tvoří virtuální 1-D mřížku $M(p)$
- **blokové:** proužky tvoří blok
- **cyklické:** řádky se střídají po 1
- **blokově-cyklické:** střídání po k řádcích
- přemapování = **AAS** operace



Mapování matic na procesy – šachovnicové:

- procesy tvoří virtuální 2-D mřížku $M(\sqrt{p}, \sqrt{p})$, přemapování opět **AAS** operace
- **blokové:** procesor má blok matice velikosti n/p
- **cyklické:** po jednotlivých prvcích matice
- **blokově-cyklické:** rozděleno na n/k bloků, procesory se střídají po blocích

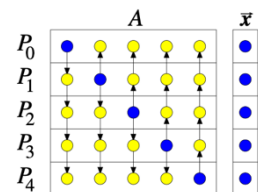


Násobení matice vektorem (MVM):

- uvažujeme matici A o velikosti $\sqrt{N} \times \sqrt{N}$ a vektor $x \sqrt{N} \times 1$, počítáme $y = Ax$
- uvažujeme v tomto případě **hustou** matici (*řádká matice = COO / CSR, otázka 14*)
- I/O-bound výpočet: každé číslo beru z matice jen 1x
- => složitost dána propustností paměťového systému (*to zde nebudeme řešit*)

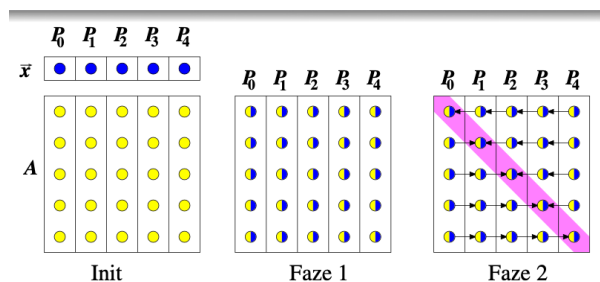
Násobení při řádkově-blokovém mapování:

- **AAB:** každý procesor pošle svou část vektorům ostatním, ty provedou pronásobení a uloží výsledek do své části vektoru y ($y_i = A_i \cdot x$ – *ideální*)
- **složitost:** $\Theta(N/p)$, komunikace: $O(N)$ => **konstantní efektivnost**



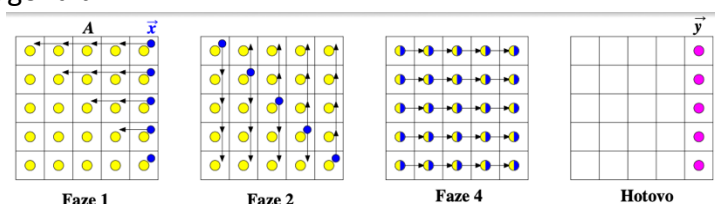
Násobení při sloupcově-blokovém mapování:

- nejprve každý procesor spočítá svůj lokální příspěvek do výsledného vektoru
- následně proběhne řádková redukce (+), výsledek pak bude na příslušné pozici diagonály (*Přijímám na svůj řádek, odesílám na jiné*)
- **složitost:** stejná u řádkově-blokového = $\Theta(N/p)$



Násobení při šachovnicově-blokovém mapování:

- vektor můžu namapovat libovolně (*například poslední sloupec*)
- krajní proces pošle příslušnou část vektoru na diagonálu
- diagonála odešle tuto část nahoru/dolů (**OAB**)
- provedeme lokální vynásobení
- řádková redukce doprava => tam výsledek
- **složitost:** stejná = $\Theta(N/p)$ a komunikace $O(N)$



49. Násobení hustých matic při šachovnicovém mapování: Cannonův a Foxův algoritmus, MPI implementace

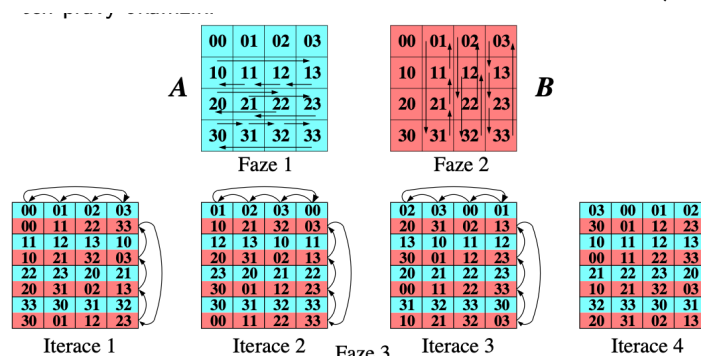
Násobení hustých matic: předpokládám klasický školní algoritmus na násobení matic
 - předpokládám blokově-šachovnicové mapování matic

Naivní algoritmus: každý procesor potřebuje odpovídající submatice pomocí **AAG**

- na závěr se provede lokální vynásobení, časová náročnost: $\Theta(N/p \cdot (\sqrt{p} + \sqrt{N}))$
- paměťově **neefektivní** (*nevleze se to do paměti jednoho procesoru*)

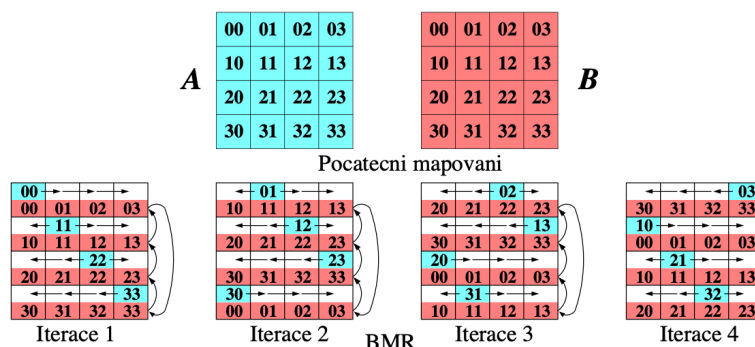
Cannonův systolický algoritmus:

- přesouvá iterativně a synchronně submatice tak, že vždy můžu násobit $A_i \cdot B_j$
- systematicky rotuji i . sloupec a j . řádek o i/j pozic pomocí **cyklický posun** `MPI_Sendrecv`
- vždy přičtu výsledek a orotuji o 1 víc, na vhodné topologii (*všeportová WH $Q_{\log p}$*) současně
- výsledná složitost: $T_{\text{Cannon}}(N, p) = O(t_s \sqrt{p}) + O\left(\frac{N}{\sqrt{p}} t_m\right) + O\left(\frac{\sqrt{N^3}}{p}\right)$



Foxův algoritmus – Broadcast-Multiply-Roll:

- nejprve se submatice pošle všem procesorům v rámci řádku i (**OAB: MPI_Bcast**)
- následně se provede lokální násobení přijatých submatic
- na závěr se provede rotace ve sloupci k o jednu pozici nahoru (**cyklický posun**)
- časová složitost, škálovatelnost podobná jako u Cannonova algoritmu



50. Mocninná metoda a její implementace v MPI: náhodné a řádkové mapování matice

Mocninná metoda: hledá iterativně největší vlastní číslo, vhodné pro velmi řídkou matici

- využití: Google PageRank

Algoritmus:

1. vytvořím nenulový počáteční vektor (*typicky* $x = (1, 1, 1, 1, \dots)$)
2. vynásobím A vektorem x , vznikne vektor $y = Ax$ (*nějaký algoritmus pro řídkou MVM*)
3. spočteme normu α vektoru y , nahradíme x normalizovaným $y = x/\alpha$ (*paralelní redukce*)
4. vyhodnotíme kritérium konvergence, pokud není splněno, pokračujeme dál

Implementace v MPI:

- předpokládáme řídkou matici, předem neurčená struktura
- procesory provádí lokální násobení, dílčí výsledky redukují (`MPI_Allreduce`)

Náhodné mapování matice:

- každý procesor potřebuje celý vektor x a vytvoří libovolný prvek vektoru y
- po provedení algoritmu má každý proces celý vektor y a α
- **složitost:** $O(n)$ paměť, kde $n = \sqrt{N}$

Řádkové mapování matice:

- každý procesor potřebuje celý vektor x , ale vektor y si již můžou rovnoměrně rozdělit
- matice rozdělena do p horizontálních pásů velikosti n/p
- získání vektoru x , složení vektoru y : `MPI_Allgather`
- rychlejší (*nepotřebujeme kopírovat y do x*), **složitost:** $x - O(n)$, $y - O(n/p)$

51. Mocninná metoda a její implementace v MPI: šachovnicové mapování matice a rozdělení komunikátorů

Šachovnicové mapování matice:

- procesy tvoří virtuální 2D mřížku $M(n,n)$
- každý procesor potřebuje jen část vektoru x (*menší paměťové nároky*)
- po lokálních MVM mají procesy příspěvek k části y a provedeme redukci
- nejpřirozenější mapování na **diagonální** procesy
- **složitost**: $x - O(n/p)$, $y - O(n/p)$

Šachovnicové mapování – rozdělení komunikátorů:

- potřebujeme provést paralelní redukci jen ve virtuálních řádcích matice procesů
- MPI: `MPI_Comm_split` – rozdělí komunikátor podle pole „barev“ (*řádková souřadnice*)
- redukci tak můžeme provádět ve všech řádcích **nezávisle** na sobě

```
MPI_Comm_split(MPI_Comm, int color, int new_rank, MPI_Comm *newComm)
```

- potřebujeme ale taky komunikátor pro **diagonální procesy**
- nejefektivnější časově i paměťově

52. Paralelní generování náhodných permutací v MPI

Generování náhodných permutací:

- chceme vygenerovat permutaci čísel 1 až n za pomoci p procesů
- na začátku deterministicky vygenerujeme posloupnost 1 až n
- **algoritmus**: náhodně promíchat přes všechny procesy

Naivní algoritmus:

1. každý proces pošle svoje číslo náhodně zvolenému procesu
 2. proces přijatá čísla lokálně náhodně promíchá
 3. procesy si navzájem přemístí čísla tak, aby měl každý n/p čísel
- extrémně **neefektivní**, hrozí zablokování (*všichni odesílatelé i příjemci*), nevíme kolik zpráv

Vylepšení:

- pamatujeme si pouze číslo náhodně zvoleného cílového procesu
- 1. čísla pro stejný cílový proces jsou sdružena a odeslána jedinou operací **AAS**
- dojde k odeslání stejného množství dat: `MPI_Alltoallv`
- je zde nutnost si předpočítat počet odesílaných dat pomocí **AAB/AAG**
- 2. lokální náhodné promíchání čísel
- 3. vyvážení výstupní permutace napříč procesy tak, aby měl každý n/p čísel