IB109 Návrh a implementace paralelních systémů

Organizace kurzu a úvod

Jiří Barnat

Organizace kurzu

Organizace kurzu

Místo a čas

- Středa 16:00-17:40, A217

Ukončení předmětu

- Závěrečný písemný test na odpřednášený obsah
- Možno získat několik bodů za nepovinné domácí úlohy
- Požadavky na úspěšné ukončení předmětu

  Z,K: bodové hodnocení testu nad 50%

  ZK: bodové hodnocení testu nad
  60%(E), 67%(D), 75%(C), 82%(B), 90%(A)

Cíl kurzu IB109

Cílem předmětu je seznámit studenty s

- Problematikou programování paralelních aplikací.
- Programátorskými prostředky pro vývoj paralelních aplikací.
- Možnostmi studia tématu na FI.

Úspěšný absolvent kurzu

- Umí identifikovat paralelně proveditelné úlohy.
- Má základní přehled o problémech souvisejících s paralelizací.
- Nebojí se implementovat vlastní vícevláknové nebo jinak paralelní aplikace či systémy.
- Má představu o tom, co se děje v zákulisí použitých knihoven pro podporu programování paralelních aplikací.
- Umí tyto knihovny správně použít.

Organizace kurzu

Osnova:

1. 21/02 — Úvod, organizace kurzu, motivace
2. 28/02 — Práce s daty ve sdílené paměti
3. 07/03 — **Zrušeno**
4. 14/03 — POSIX Threads 1
5. 21/03 — POSIX Threads 2
6. 28/03 — Lock-Free programování
7. 04/04 — OpenMP, TBB, C++11
8. 11/04 — Principy návrhu paralelních algoritmů
9. 18/04 — **Zrušeno**
10. 25/04 — Předávání zpráv v distribuované paměti, MPI
11. 02/05 — Kolektivní komunikace
12. 09/05 — Složitostní analýza paralelních programů
13. 16/05 — **Zkouška – 1. termín**

Předpoklady a kontext na FI

IB109

- Úvodní kurz určený pro bakalářské studium.
- Povinný v rámci oboru *Paralelní a distribuované systémy*

Předpoklady

- Základní znalosti o fungování výpočetních prostředků a operačních systémů.
- Základní zkušenost s imperativním programováním sekvenčních algoritmů.

Kontext v rámci FI

PV192 – Paralelní algoritmy

- Paralelní zpracování, Klasifikace paralelních systémů, Úrovně paralelismu, Paralelní počítače, Systémy s distribuovanou pamětí, MPI

PV197 – GPU Programming

- Paralelní výpočty na grafických kartách s technologií CUDA.

PA150 – Principy operačních systémů

- Vlákna, procesy, monitory, semafory, synchronizace.
- Hierarchie pamětí.

IA039 – Architektura superpočítačů a intenzivní výpočty

- Procesory. Paralelní počítače. Překladače. MPI. PVM a koordinační jazyky. Profilování a měření výkonu.

Kontext v rámci FI – pokr.

IV100 – Paralelní a distribuované výpočty

- Distribuované systémy a algoritmy. Komunikační protokoly. Směrovací algoritmy a tabulky. Distribuované algoritmy pro detekci ukončení, volbu vůdce, vzájemné vyloučení, hledání nejkratší cesty. Byzantská shoda.

IV010 – Komunikace a paralelismus

- Teoretický model paralelních procesů a komunikace. CCS. Synchronizace, vnitřní akce. Ekvivalence systémů pomocí slabé/silné bisimulace a relace kongruence.

IV113/IA169 – Úvod do validace a verifikace

- Model checking, verifikace paralelních programů.

Kontext v rámci FI

Laboratoře na FI

- SITOLA
- PARADISE
- SYBILA

Projekty

- IV112 – Projekt z programování paralelních aplikací
- PV197 – GPU Programming

Studijní materiály

Knihy

- Maurice Herlihy, Nir Shavit: *The Art of Multiprocessor Programming*
- A. Grama, A. Gupta, G. Karypis, V. Kumar: *Introduction to Parallel Computing*
- I. Foster: *Designing and Building Parallel Programs*
- W. Group, E. Lusk, A. Skjellum: *Using MPI*
- ...

Studijní materiály

E-zdroje:

- http://www.wikipedia.org
- Kurzy a jejich studijní materiály na různých univerzitách
  - http://www.cs.arizona.edu/people/greg/mpdbook (Foundations of Multithreded, Parallel, and Distributed Programming)
  - http://renoir.csc.ncsu.edu/CSC495A

- http://www.hlrs.de/organization/par/par_prog_ws Parallel Programing Workshop (MPI, OpenMP)
- Domovské stránky projektů MPI, TBB, OpenMP, ...
- Online tutoriály
- ...

Motivace

Souběžnost a Výpočetní paralelismus

Souběžnost

- Existence dvou a více procesů (v obecném smyslu slova) v jeden časový okamžik.

Výpočetní paralelismus

- Napříč všemi úrovněmi, od implementace registru až po koexistenci rozsáhlých výpočetně distribuovaných aplikací.
- Chtěný pro zvýšení výkonu.
- Nutný kvůli prostorové distribuci.
- Vykoupený složitostí návrhu a pořizovací cenou.

Souběžnost v Computer Science

- Specifikace, implementace a analýza paralelních a distribuovaných systémů.
- Inherentně sekvenční algoritmy, složitostní třída *NC*.

Důvody vývoje paralelních aplikací

Výkonnost

- Umožní efektivně využít aggregované výpočetní prostředky k zrychlení výpočtu.

Proveditelnost

- Agregace výpočetní síly není volba, ale nutnost pro dokončení úlohy (velký objem výpočtů, nepřijatelná latence).

Bezpečnost

- Duplikace klíčových částí systému pro případ havárie, či ohrožení důvěry jedné části.

Cena

- Oddělení nesouvisejících částí aplikace, levnější údržba.
- Agregovaná výpočetní síla je levnější.

Výpočetních platformy – abstraktně

Abstraktní model výpočetního systému

Procesor - Datová cesta - Datové úložiště

- Všechny části systému mohou být úzkým místem vůči výkonnosti aplikace jako celku.
- Paralelismus je přirozený způsob překonání úzkých míst.

Procesory

Procesory

- Neustálá potřeba zvyšovat výkon.
- Výkon procesoru spojován s Moorovým zákonem.

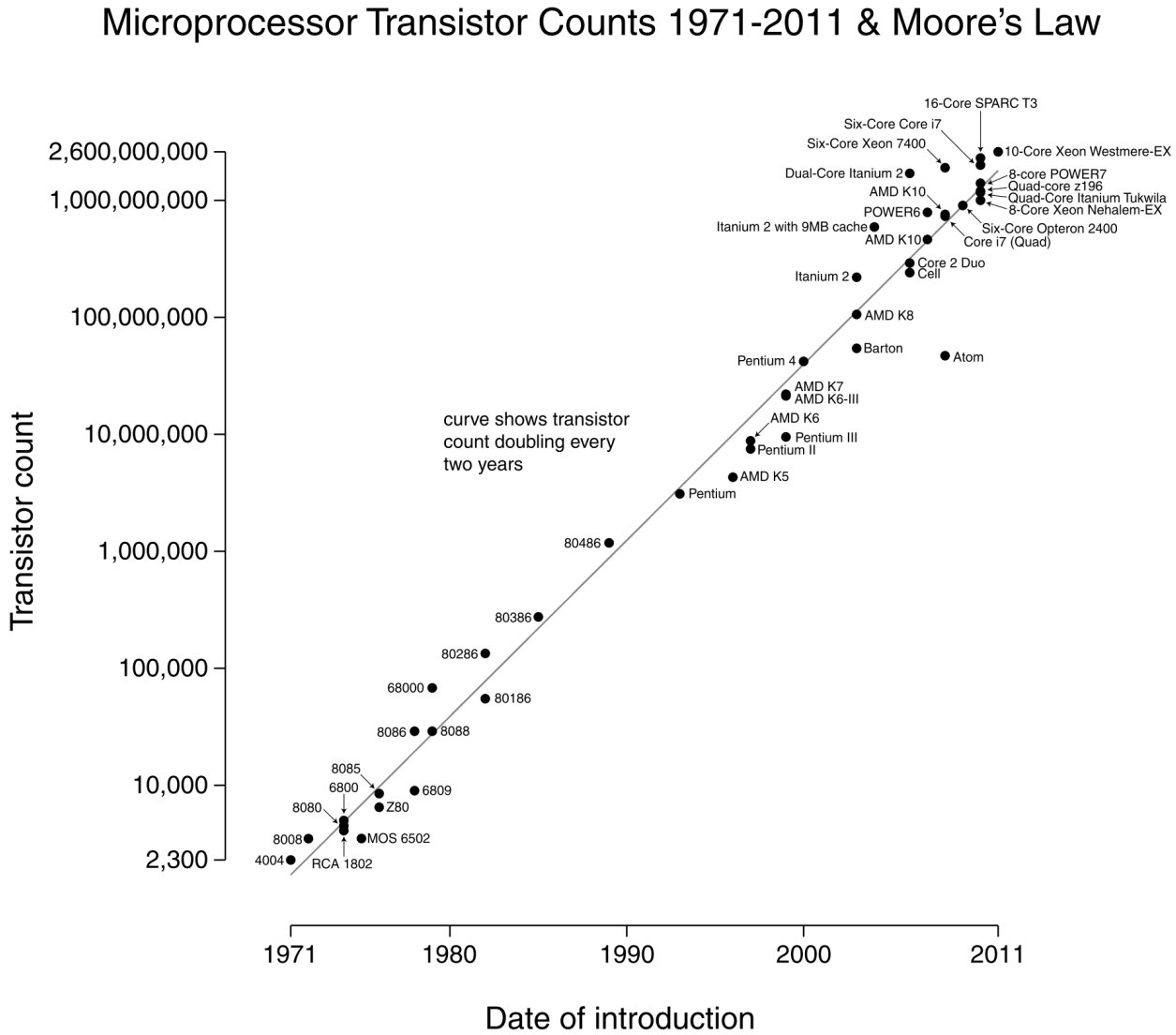
Moorův zákon

- Gordon Moore, spoluzakladatel Intelu
- Počet tranzistorů v procesoru se zdvojnásobí přibližně každých 18 měsíců.

Metody zvyšování výkonu procesorů

- Zvyšování frekvence vnitřních hodin.
- Multiplicita, Paralelismus

Moore's Law


Microprocessor Transistor Counts 1971-2011 & Moore's Law
Transistor count
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
curve shows transistor count doubling every two years
16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
AMD K10
Quad-Core Itanium Tukwila
POWER6
8-Core Xeon Nehalem-EX
Itanium 2 with 9MB cache
Six-Core Opteron 2400
AMD K10
Core i7 (Quad)
Core 2 Duo
Itanium 2
Cell
AMD K8
Barton
Pentium 4
Atom
AMD K7
AMD K6-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
8080
Z80
8008
MOS 6502
4004
RCA 1802
1971
1980
1990
2000
2011
Date of introduction

Trend ve vývoji procesorů

Pozorování

- Výrobcům procesorů se nedaří zvyšovat výkon jednoho jádra.
- Fyzikální zákony brání neustálé miniaturizaci – okolo 5nm již nelze udržet elektrony v atomu.
- Současná technologie 14-16nm (dříve 22nm, 28nm, 32nm, 45nm, 65nm, 90nm).

Řešení

- Multi-core a many-core procesory.
- Pravděpodobný způsob zvyšování výkonu i v budoucnosti.
- Částečný odklon od jednotek monolitických jader k desítkám menších specializovaných, či k hybridním řešením.

Důsledek

- Sekvenční algoritmy nemohou nadále těžit z rostoucího výkonu procesorů.
- Paralelizace výpočtů je nevyhnutelný směr vývoje.

Datové cesty

Role paralelismu v komunikaci

- Větší propustnost komunikačních linek s následným efektem snižování latence.
- Robustnost a spolehlivost komunikačních linek.

Příklady paralelismu na datové cestě

- Šířka sběrnice 32/64/128 bitů.
- Domácí dual-band WIFI router.

Paměť

Fakta

- Výkon procesorů převyšuje výkon ostatních komponent.
- Cesta **procesor – paměť – disk** je zdlouhavá.
- Doba nutná pro získání jednotky informace z paměti roste se vzdáleností místa uložení od místa zpracování.

Víceúrovňové uložení informací

- Registry procesoru
- L1/L2/L3 cache
- Operační paměť
- Cache I/O zařízení
- Magnetické/optické mechaniky

Cache

Cache paměť obecně:

- Kopie části dat v rychleji dostupném místě.
- Může a nemusí být kontrolovatelná uživatelem nebo OS.

Příklady Cache s různou možností kontroly

- L1/L2 cache v rámci CPU nekontrolovatelná programátorem
- I/O efficient algoritmy
  - Obcházejí virtualizaci paměti kontrolovanou OS, a místo toho realizují vlastní způsob použití operační paměti jako cache pro data na disku.

Použití mnoha paměťových modulů

Multiplicita paměťových modulů

- Větší množství uložitelných/zapamatovatelných informací.
- Větší množství linek do paměti (větší propustnost).
- Větší režie na udržení konzistence.

Příklady

- Disková pole
- Peer-to-Peer sítě
- NUMA architektury
  - Více procesorové počítače s více paměťovými moduly uspořádanými tak, že přístup jednoho procesoru do různých paměťových modulů je různě rychlý.

Paralelní výpočty

Paralelismus z pohledu OS

Multitasking na jednom výpočetním jádru

- Aplikace se v běhu na CPU jádru střídají.
- Na jednom CPU zdánlivě "běží"více aplikací.
- Jednotka plánování OS je proces.

Multitasking na více výpočetních jádrech

- Různé aplikace přiřazeny na různá výpočetní jádra.
- Jinak standardní multitasking na každém jednom jádře.
- Jednotka plánování OS je proces.

Multitasking a multithreading

- Každá aplikace může mít více výpočetních vláken.
- Vlákna se v běhu na jednom CPU jádru střídají.
- Vlákna jedné aplikace mohou běžet na různých jádrech.
- Jednotka plánování OS je vlákno.

Flynnova klasifikace

Single Instruction Single Data

- V daný okamžik je zpracovávána jedna instrukce, která pracuje nad jedním datovým proudem.
- Klasický sekvenční výpočet.

Single Instruction Multiple Data

- Jedna tatáž instrukce je vykonávána nad více datovými proudy.
- Vektorové instrukce CPU, architektura GPU.

Multiple Instruction Multiple Data

- Nezávislý souběh dvou a více SISD, SIMD přístupů.
- Více jádrové procesory.

Multiple Instruction Single Data

- Prakticky se nevyskytuje.

Distribuovaný systém

Distribuovaný/paralelní systém

- Specifikován po částech (procesy).
- Chování systému vzniká interakcí souběžných procesů.
- Emergentní jevy.

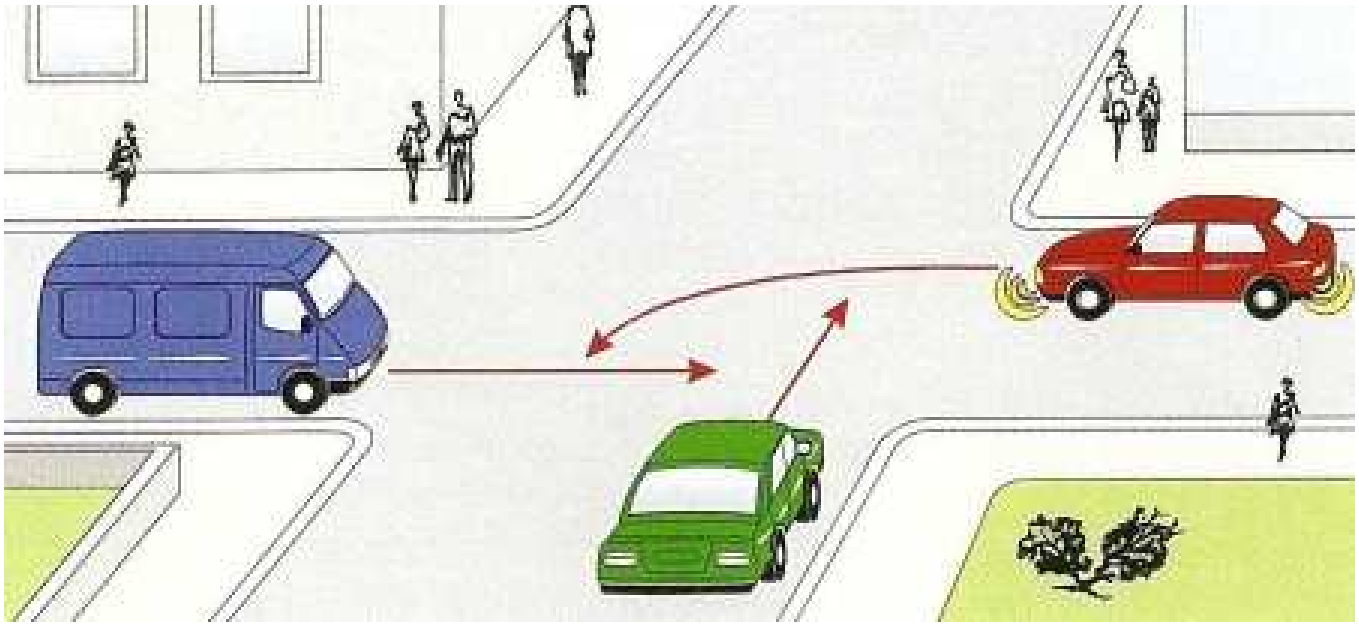
Synchronizace

- Omezení na prokládání a souběh akcí jednotlivých procesů distribuovaného systému.

Komunikace

- Přenos informace z jednoho procesu na druhý.

Příklad distribuovaného systému



Vybrané problémy distribuovaných systémů

Jevy

- Nekonzistentní vize konzistentního světa.
- Vzájemná interference

Rizika

- Race-condition, nedeterministické chování.
- Uváznutí (Deadlock, Livelock)
- Stárnutí, hladovění (Starving)
- Přetečení buferů, problém producent-konzument.
- Zbytečná ztráta výkonu (aktivní čekání).

Vývoj paralelních systémů je náročnější

Důvody:

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

Příklad

- V minulých letech byl doporučován pro hry 2-jádrový procesor, proč ne 4-jádrový, když byl zcela určitě výkonnější?

Vývoj paralelních systémů je náročnější

Důvody:

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

Příklad

- V minulých letech byl doporučován pro hry 2-jádrový procesor, proč ne 4-jádrový, když byl zcela určitě výkonnější?
- Je obtížné napsat herní engine, který by fungoval dobře na 2-jádrovém stroji a na 4-jádrovém stroji běžel 2x rychleji, je preferována vyšší frekvence 2-jádrového procesoru.

HPC

HPC

HPC (High Performance Computing)

- Oblast Computer Science
- Výpočty na vysoce paralelních platformách

Nejvýkonější počítač světa [Jaro 2018]

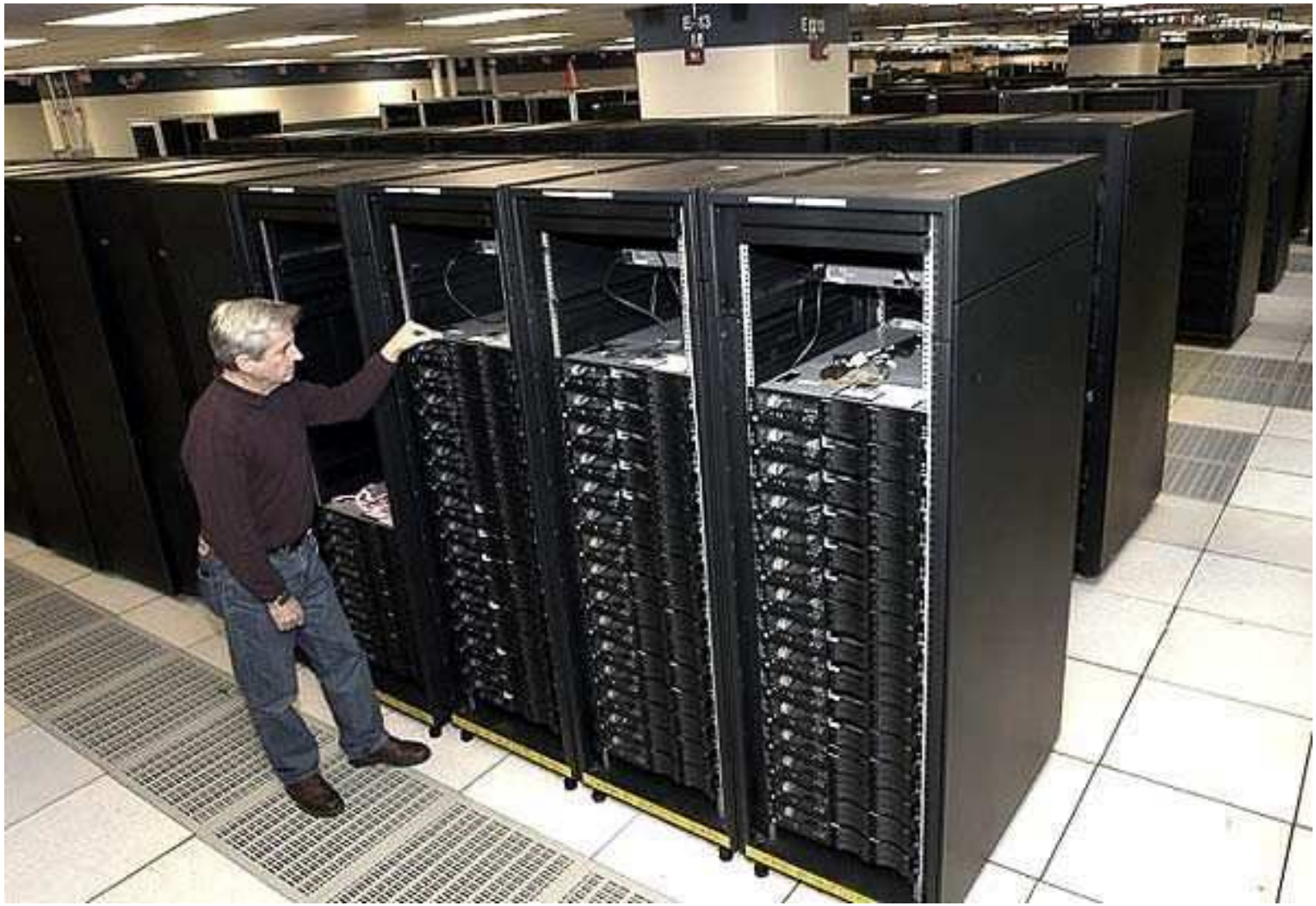
- National Supercomputing Center in Wuxi
- TFLOPS: 93 014

Nejvýkonější počítač světa [Jaro 2010]

- Roadrunner
- TFLOPS: 1 105

- Více viz www.top500.org.

Nejvýkonnější počítače

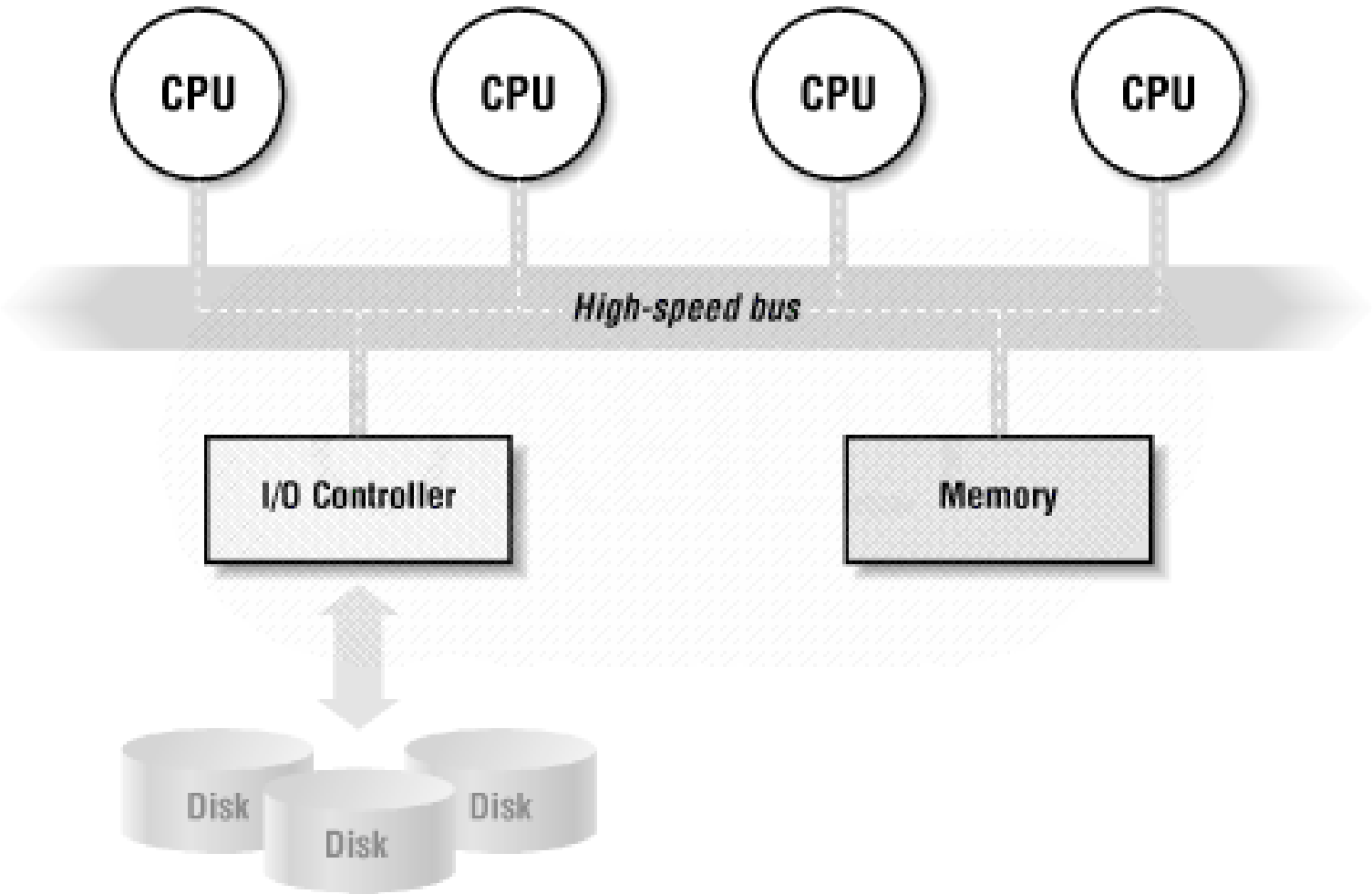


IB109 Návrh a implementace paralelních systémů

Programování v prostředí se sdílenou pamětí

Jiří Barnat

HW model prostředí se sdílenou pamětí



HW platformy

Paralelní systémy se sdílenou pamětí

- Systémy s více procesory
- Systémy s více-jadernými procesory
- Systémy s procesory se zabudovaným SMT
- Kombinace

Rizika paralelních výpočtů na soudobých procesorech

- Mnohé optimalizace na úrovni procesoru byly navrženy tak, aby zachovávaly sémantiku sekvenčních programů.

Pozor zejména na

- Přeuspořádání instrukcí
- Odložené zápisy do paměti

Simultání multithreading (SMT)

Princip

- Procesor využívá prázdné cykly způsobené latencí paměti k vykonávání instrukcí jiného vlákna.
- Vyžaduje duplikaci jistých částí procesoru (např. registry).
- Vlákna sdílí cache.

Příklad: Intel Pentium 4

- Hyper-Threading Technology (HTT)
- OS s podporou SMP vidí systém se SMT/HTT jako více procesorový systém.
- Až 30% nárůst výkonu, ale vzhledem ke sdílené cache může být rychlost výpočtu jednoho vlákna nižší.

Více-jaderné procesory (multicores)

Více plnohodnotných procesorů v jednom chipu.

Výhody

- Efektivnější cache koherence na nejnižší úrovni.
- Nižší náklady pro koncového uživatele.

Nevýhody

- Víc jader emituje větší zbytkové teplo.
- Takt jednoho jádra bývá nižší.
- Automatické dočasné podtaktování/přetaktování.
- Jádra sdílí datovou cestu do paměti.

Realita

- Více-jádrové procesory se SMT.
- Intel Core-i7 (hexa-core se SMT = 12 paralelních jednotek)

Paralelismus v prostředí se sdílenou pamětí

Idealizovaný model

- **Na této úrovni se řeší návrh paralelního algoritmu.**
- Jednotlivá výpočetní jádra paralelního systému pracují zcela nezávisle.
- Přístupy k datům v paměti jsou bezčasové a vzájemně výlučné.
- Komunikace úloh probíhá atomicky přes sdílené datové struktury.

Realita

- **Na této úrovni musí programátor řešit technickou realizaci paralelního algoritmu.**
- Přístup do paměti přes sběrnici je pro CPU příliš pomalý.
- Registry procesoru a cache paměti – rychlé kopie malého množství dat na různých místech datové cesty.
- Problém koherence dat.

Paralelní úlohy v kontextu OS – Procesy a vlákna

Procesy

- Skrývají před ostatními procesy své výpočetní prostředky.
- Pro řešení paralelní úlohy je potřeba mezi-procesová komunikace (IPC).
  - Sdílené paměťové segmenty, sokety, pojmenované a nepojmenované roury.

Vlákna

- Existují v kontextu jednoho procesu.
- V rámci rodičovského procesu sdílí výpočetní prostředky.
- Komunikace probíhá přes sdílené datové struktury.
- Účelem interakce je spíše synchronizace než transport dat.
- Subjekty procedury plánování.

Příklad použití vláken v rámci procesu

Vlákno

- Realizuje výpočet, tj sekvenci instrukcí.
- Každý proces je tvořen alespoň jedním vláknem.
- Hlavní vlákno procesu vytváří další vlákna.

Příklad

```
1 for (i=0; i<n; i++)
2   for (j=0; j<n; j++)
3     m[i][j] = create_thread(
4                 product(getrow(i),getcol(j))
5               )
```

Procesy versus vlákna

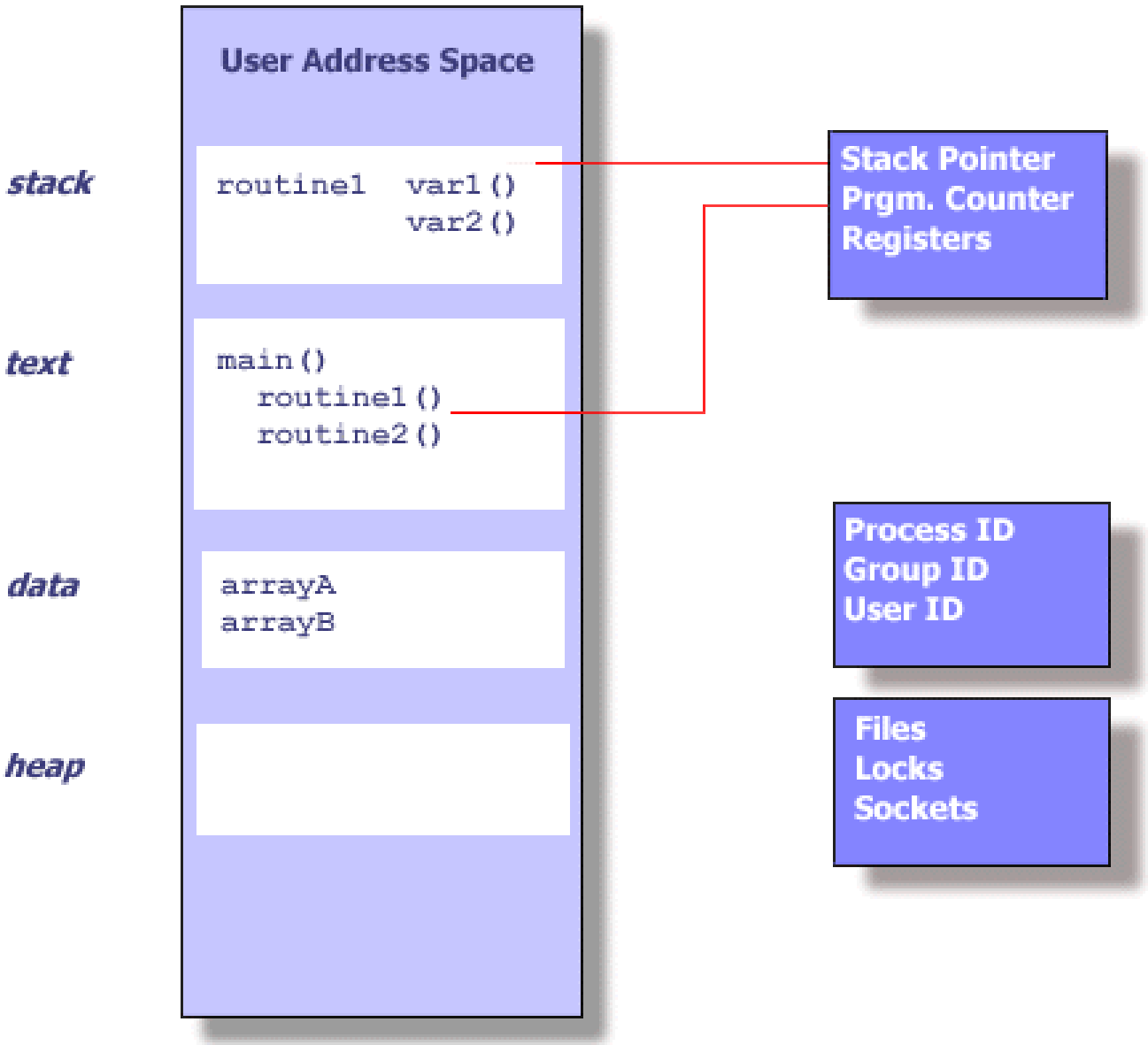
Proces

- Identifikátory procesu a vlastníka
- Proměnné prostředí, pracovní adresář
- Kód
- Registry, Zásobník, Halda
- Odkazy na otevřené soubory a sdílené knihovny
- Reakce na signály
- Kanály IPC

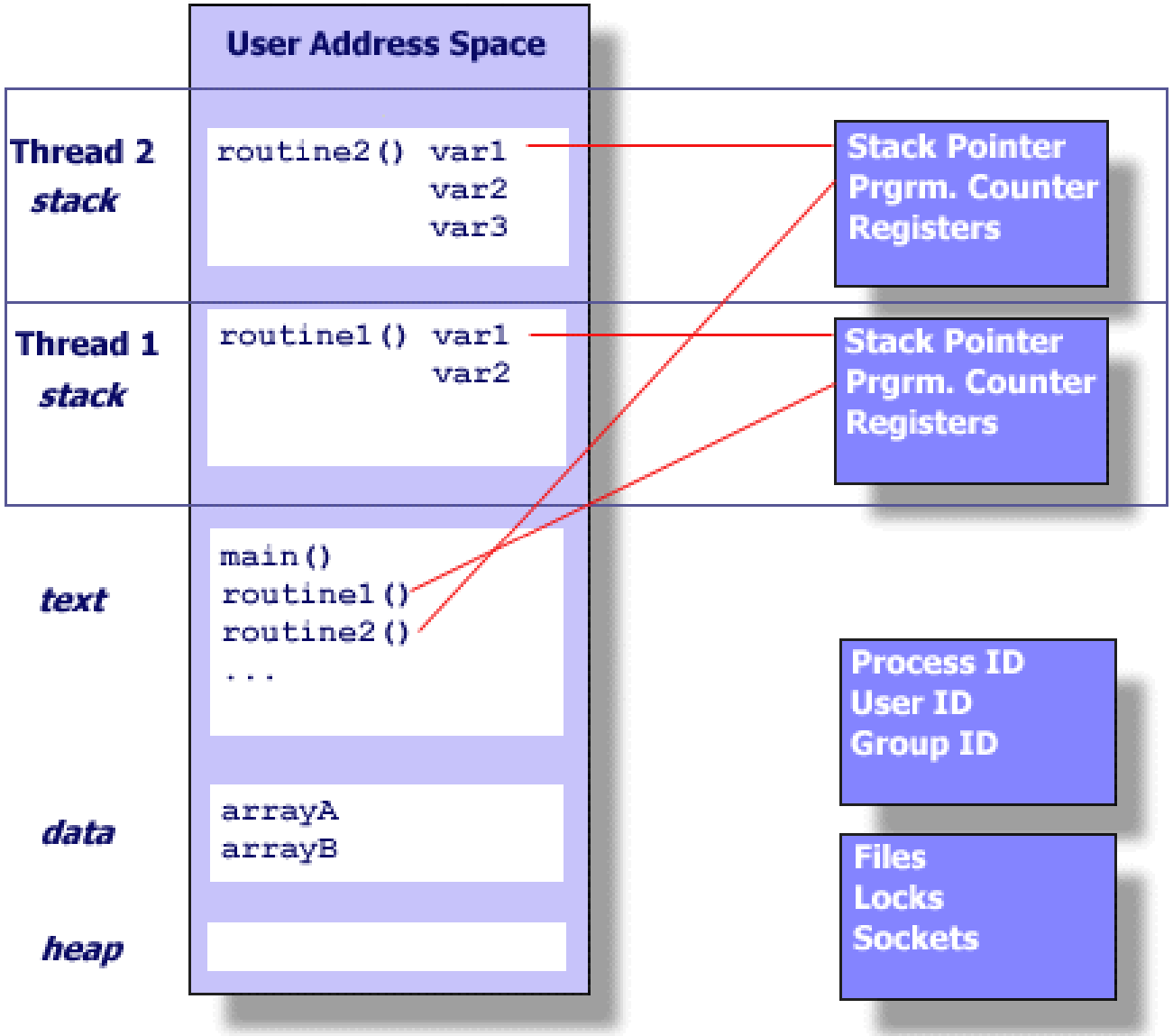
Vlákna mají privátní

- Zásobník
- Registry
- Frontu signálů

Proces v operačním systému



Vlákna v rámci procesu



Argumenty pro používání vláken místo procesů

Výkon aplikace

- Vytvoření procesu je výrazně dražší než vytvoření vlákna.
- Změna dat provedená v rámci jednoho vlákna je viditelná v kontextu celého procesu.
- Komunikace mezi vlákny spočívá v předávání reference na data, nikoliv v předávání obsahu.
- Předávání reference se děje v rámci jednoho procesu, operační systém nemusí řešit skrývání dat a přístupová práva.

Nevýhody

- Vlákna nemají žádné "soukromí".
- Sdílené globální proměnné.

Efektivní využití cache

Cache

Princip cache

- Menší ale rychlejší paměť na pomalé datové cestě.
- Při prvním čtení se okolí čtené informace uloží do cache.
- Při následujícím čtení z okolí původní informace se čte pouze z cache.

Koherence

- Soulad dat uložených v paměti počítače (potažmo v cache).
- Je zajištěno, že existuje právě jedna platná hodnota asociovaná s daným paměťovým místem.
- Z důvodu rychlosti jsou zápisy procesoru do paměti odkládány a sdružovány, bližší specifikace chování procesoru v tomto ohledu je dána **paměťovým modelem** daného CPU.

Cache – související pojmy

Související pojmy

- Cache line – atomický paměťový blok uložený v cache.
- "Hit ratio" – číslo vyjadřující úspěšnost obsloužení požadavků na data daty uloženými v cache.
- "Vylití cache" – procedura aktualizace dat v paměti hodnotami uloženými v cache.

Zásady efektivního použití cache

- Časová lokalita – přístupy v malém časovém intervalu.
- Prostorová lokalita – přístup k datům uložených adresně blízko sebe.
- Zarovnaná alokace paměti (např. memalign (GNU C)).

False sharing – příklad

Specifikace

- Paralelní cache koherentní systém s více procesory.
- Program s několikanásobně vícero vlákny.
- Pole hodnot `int pole[nr_of_threads]`.
- Vlákna počítají výslednou hodnotu typu `int` a k výpočtu si ukládají mezivýsledek typu `int`.

Varianty implementace

A) Každé vlákno opakovaně zapisuje do datového pole integerů na pozici určenou jeho ID.

B) Každé vlákno zapisuje do lokální proměnné a před skončením nakopíruje hodnotu do pole na pozici určenou jeho ID.

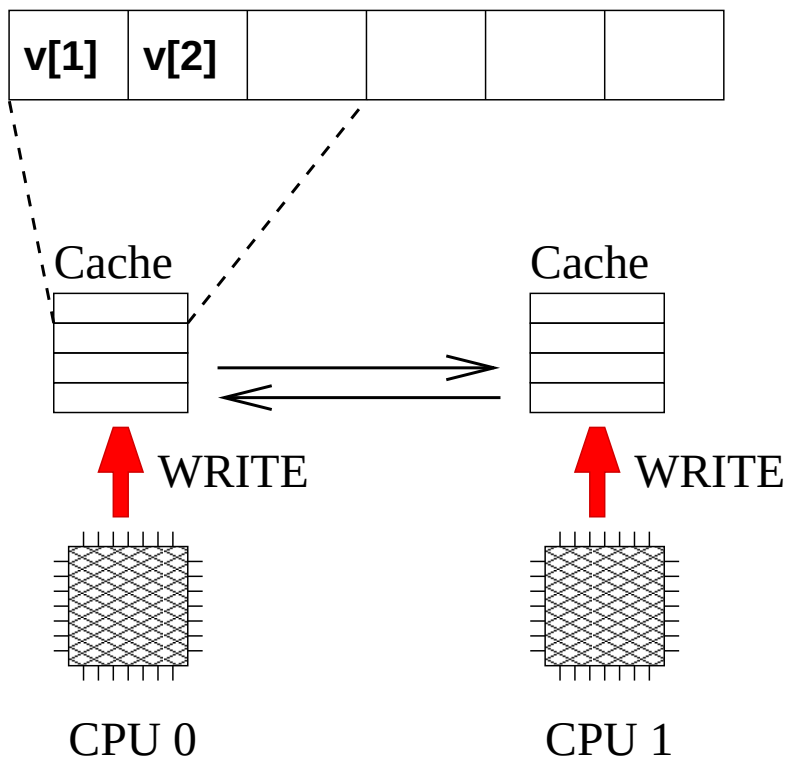
Otázka

- Která implementace bude pomalejší a proč?

False sharing

```
typedef struct {
    long long iter;

} thread_private_data_t;


thread_private_data_t v[16];
```

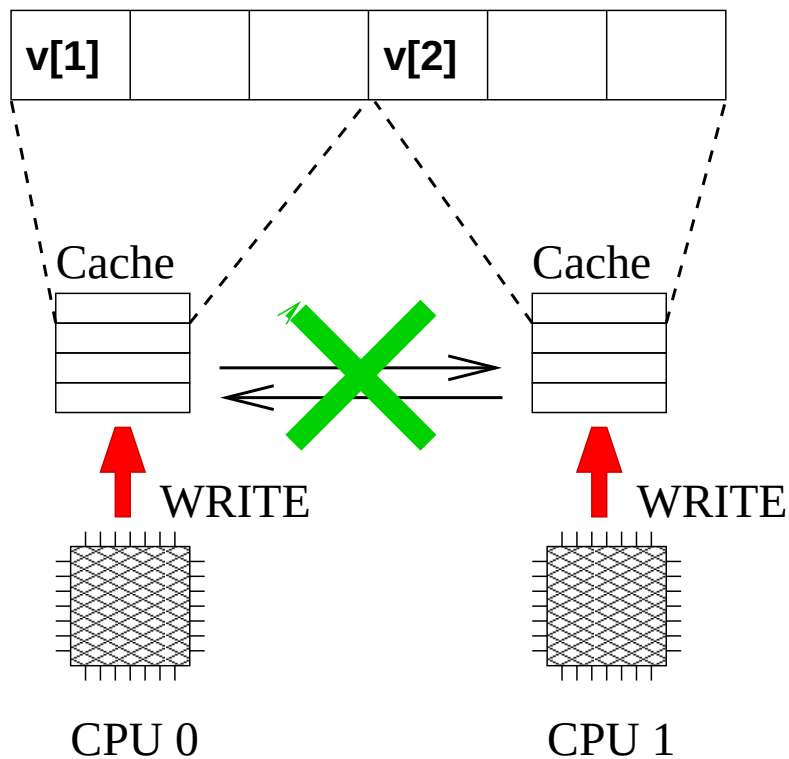
```
my_thread(...) {
    ...
    v[id].iter ++;
    ...
}
```



False sharing

```
typedef struct {
    long long iter;
    char cache_line_filler[2000];
} thread_private_data_t;


thread_private_data_t v[16];
```

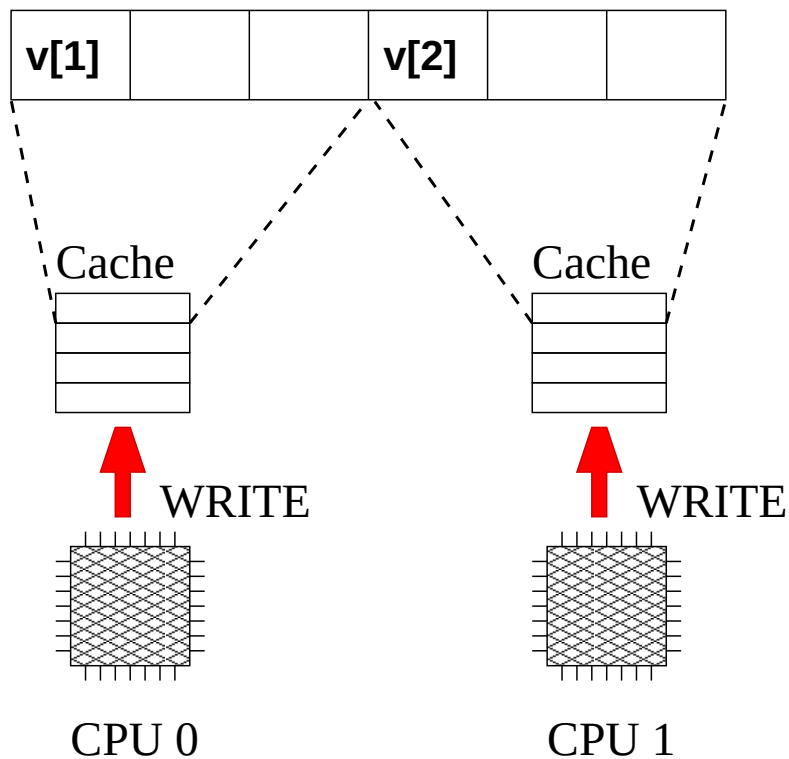
```
my_thread(...) {
    ...
    v[id].iter ++;
    ...
}
```



False sharing

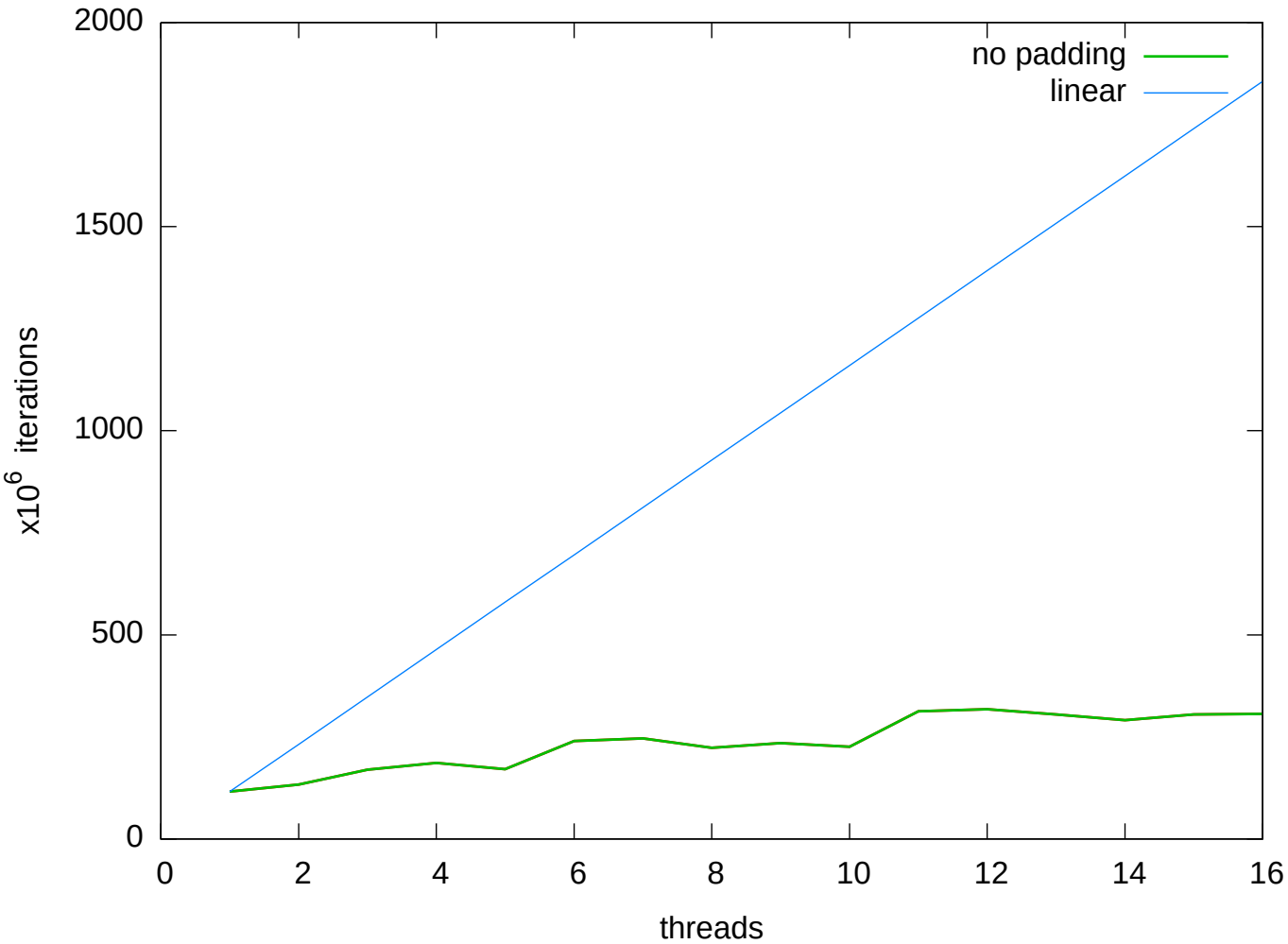
```
typedef struct {
    long long iter;
    char cache_line_filler[2000];
} thread_private_data_t;


thread_private_data_t v[16];
```

```
my_thread(...) {
    ...
    v[id].iter ++;
    ...
}
```



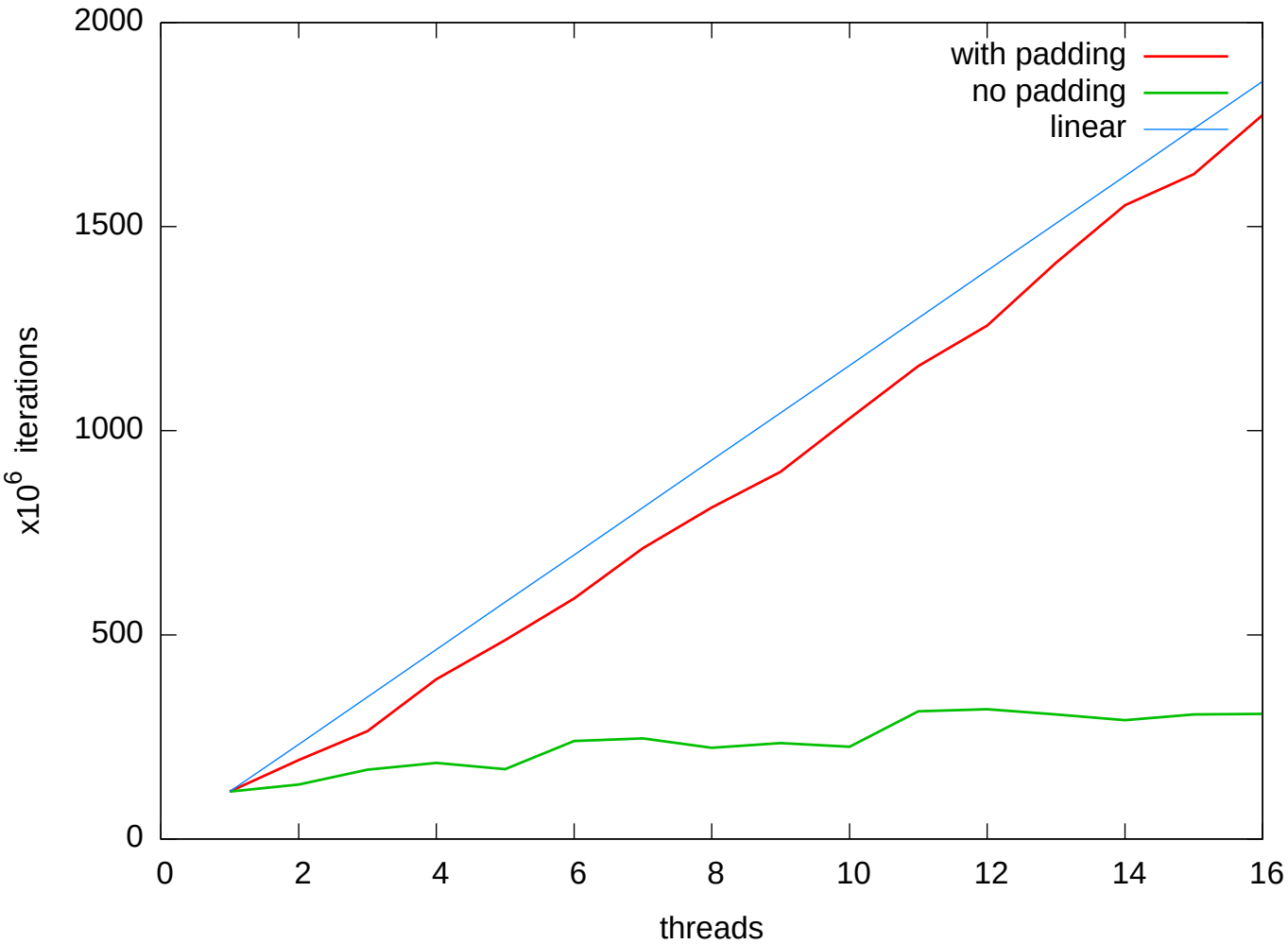
False sharing

1 thread = 119 $x10^6$ iterations (1.00x)
10 threads = 231 $x10^6$ iterations (1.94x)
16 threads = 313 $x10^6$ iterations (2.63x)



False sharing

1 thread = 119 $x10^6$ iterations (1.00x)
10 threads = 1055 $x10^6$ iterations (8.86x)
16 threads = 1815 $x10^6$ iterations (15.25x)



Přehnaná důvěra v mentální model

POZOR!

- Není garantováno, že všechny informace uložené do explicitně definovaných proměnných budou zapsány do paměti.
- Proměnné mohou být realizovány registrem procesoru.

Důsledek

- Posloupnost hodnot uložených do jedné proměnné v rámci jednoho vlákna může být viděna jiným vláknem jen **částečně** nebo **vůbec**.

Nestálé proměnné

Pozorování

- Udržování koherence cache pamětí je nákladné.
- Soudobé překladače nevynucují, aby každá vypočítaná hodnota byla uložena do paměti (nevygenerují instrukci ukládající obsah registru do paměti).

Důsledek

- Modifikace sdílené proměnné provedená v jednom vlákně na daném CPU se nemusí projevit v jiném vlákně na jiném CPU.

Příklad

```
int i=0;
```

```
P0 {
  while ( i==0 );
}
```

```
P1 {
  if (i==0) i++;
}
```

Nestálé proměnné

Problém

- Vlákno P0 neskončí, neboť nezaznamená změnu proměnné `i`.
- Může záviset na stupni optimalizace překladu, například
  - chyba se neprojeví při překladu pomocí `g++ -O0`
  - chyba se projeví při překladu pomocí `g++ -O2`

Řešení

- Je nutné označit proměnnou jako tzv. nestálou proměnnou.
- C,C++: klíčové slovo `volatile`.
- Překladač zajistí, aby daná proměnná nebyla realizována pouze na úrovni registrů CPU, ale před a po každém použití byla načtena/uložena do paměti.

Příklad

```
volatile int i=0;

P0 {                          P1 {
  while ( i==0 );               if (i==0) i++;
}                             }
```

Nestálé proměnné – volatile

Použití klíčového slova

- Umístěno před nebo za datový typ v definici proměnné.
- Rozlišujeme
  - nestálou proměnnou:
    ```
    volatile T a
    T volatile a
    ```
  - ukazatel na nestálou proměnnou:
    ```
    volatile T *a
    ```
  - nestálý ukazatel na nestálou proměnnou:
    ```
    volatile T * volatile a
    ```

Případy, kdy je nutné použít volatile:

- Proměnná sdílená mezi souběžnými vlákny/procesy.
- Proměnná zastupující vstup/výstupní port.
- Proměnná modifikovaná procedurou obsluhující přerušení.

Přehnaná důvěra ve volatile

Příklad

```
volatile int i=0;

Proc {                          main {
  for (int j=0;                   run_thread Proc as T1;
       j<100000;                  run_thread Proc as T2;
       j++)                       wait_on T1;
    i=i+1;                        wait_on T2;
}                                 print i;
                                }
```

Přehnaná důvěra ve volatile – pokračování

Výstup

- Na systému s jednou výkonnou jednotkou výstup vždy 200000.
- Na paralelních architekturách výstup často menší než 200000.

Zdůvodnění

- Přičtení není realizováno jednou instrukcí, riziko proložení vláken.
- Zápisy hodnot do paměti (do cache) nejsou prováděny v okamžiku zpracování instrukce, ale jsou odkládány a shlukovány.
- Přesný popis toho, jak procesor manipuluje se zápisy do paměti je součástí specifikace procesoru, jedná se o tzv. **paměťový model**.

Přehnaná důvěra v mentální model

POZOR!

- Pořadí zápisů do paměti dle vykonávané posloupnosti instrukcí procesoru nemusí korespondovat se skutečným pořadím zápisu hodnot do paměti.
- Paměťový model procesoru garantuje korektnost pouze pro sekvenční programy.

Důsledek

- Posloupnost přiřazení hodnot různým sdíleným proměnným provedená v jednom vlákně nemusí korespondovat s pořadím změn hodnot těchto proměnných v jiném vlákně.

Největší "skrčka" paralelního programování

Příklad

```
volatile int i=0;
volatile int * volatile p=0;

P0 {                          P1 {
  ...                           ...
  while (i==0);                 p = new int;
  (*p)=5;                       i=1;
  ...                           ...
}                             }
```

Problém

- V okamžiku přístupu na adresu odkazovanou ukazatelem `p` může mít `p` hodnotu 0.

Pozorování

- Bez nějakého dalšího opěrného bodu na HW úrovni je programování paralelních systémů téměř nemožné.

Paměťová bariéra

Co je to

- HW primitivum pro synchronizaci stavu paměti a stavů procesorů v daném místě programu.
- Na soudobých procesorech realizované instrukcí `mfence`.

Přesný popis

- Na hardwarové úrovni provede serializaci všech *load* a *store* instrukcí, které se vyskytují před instrukcí `mfence`. Tato serializace zajistí, že efekt všech instrukcí před instrukcí `mfence` bude globálně viditelný pro všechny instrukce následující za instrukcí `mfence`.

Realizace

- Není součástí vyšších programovacích jazyků.
- Různé překladače dávají programátorovi jisté možnosti.
  - GCC: `__sync_synchronize()`
  - Intel(R) C++ Compiler: `void_mm_mfence(void)`

HW podpora pro atomické instrukce

Fakta

- Paměťová bariéra neřeší problém atomických instrukcí jako jsou TEST-AND-SET, COMPARE-AND-SWAP, atd.
- Instrukce výše zmíněného typu jsou však pro účely efektivního paralelního programování velmi vhodné.

Další HW podpora

- Alpha, Mips, PowerPC, ARM: instrukce typu LL/SC
- x86 architektura
  - `lock` – následující (do paměti zapisující) instrukce proběhne atomicky a její efekt bude ihned globálně viditelný
  - `XCHG` – prohodí obsah registru a paměťového místa (obsahuje z definice prefix `lock`)

Realizace atomických instrukcí na úrovni kódu

Možnost 1: Jazyk symbolických adres

```
1 int test_and_set(volatile int *s){
2   int r;
3   __asm__ __volatile__(
4     "xchgl %0, %1 \n\t"
5     : "=r"(r), "m"(*s)
6     : "0"(1), "m"(*s)
7     : "memory");          ← paměťová bariéra
8  return r;}
```

Možnost 2: Zabudované funkce překladače (GCC $\geq$ 4.1)

- `type __sync_val_compare_and_swap (...)`
- `type __sync_fetch_and_add (...)`

Možnost 3: Součást programovacího jazyka

- C++ rev. 11, Java, ...

IB109 Návrh a implementace paralelních systémů

Programování v prostředí se sdílenou pamětí

Jiří Barnat

Rizika spojená se sdílenou pamětí

Race condition

Pozorování

- Paralelní programy mohou při opakovaném spouštění zdánlivě náhodně vykazovat různá chování.
- Výsledek provedení programu může záviset na absolutním pořadí provedení instrukcí programu, tj. na proložení instrukcí zúčastněných procesů/vláken.

Race condition

- Nedokonalost paralelního programu, která se projevuje takovýmto nedeterministickým chováním se označuje jako **race condition**, (zkráceně race).

Race condition – příklad

Příklad

```
*myStructure p;

P0 {
  p = new myStructure;
  p -> data = 1;
  cout «(p->data)«endl;
}
```

```
P1 {
  p = new myStructure;
  p -> data = 2;
  cout «(p->data)«endl;
}
```

Atomicita operací

Pozorování

- Jednoduchý příkaz ve vyšším programovacím jazyce neodpovídá nutně jedné instrukci procesoru.
- V moderních operačních systémech je každé vlákno podrobeno plánovacímu procesu.
- Vykonání posloupnosti instrukcí procesoru odpovídající jednomu příkazu vyššího programovacího jazyka může být přerušeno a proloženo vykonáním instrukcí jiného vlákna.

Atomicita operací

Příklad

- Přičtení čísla do proměnné efektivně může znamenat načtení proměnné do registru, provedení aritmetické operace, a uložení výsledku do paměti.
- Při vhodném souběhu následujících procesů, se může efekt jednoho přiřazení do sdílené globální proměnné zcela vytratit

```
volatile int a=0;

P0 {                    P1 {
  a = a + 10;             a = a + 20;
}                       }
```

- Demonstrujte proložení instrukcí, které vyústí v jinou hodnotu, než 30.

Relativní rychlost výpočtu

Pozorování

- Nelze spoléhat na současný souběh vláken, potažmo relativní rychlost výpočtu jednotlivých vláken.

Příklad

```
volatile int a=0;

    P0 {                    P1 {
      usleep 200;             a = 1;
      a = 0;                  usleep 200;
    }                       }
```

- Po skončení obou vláken (současně spuštěných) bude mít sdílená proměnná ve většině případů hodnotu 0. Není to však ničím garantováno, tj. může nastat situace, kdy bude mít hodnotu 1.

Uváznutí

Uváznutí (Deadlock)

- Pokud mají vlákna inkrementální požadavky na unikátní sdílené zdroje, může dojít k tzv. uváznutí, tj. nemožnosti pokračování ve výpočtu.

Příklad

```
P0 {                    P1 {
  zamkni A;               zamkni B;
  zamkni B;               zamkni A;
...                     ...
}                       }
```

Hladovění

Hladovění, Stárnutí, Neprogrese (Livelock)

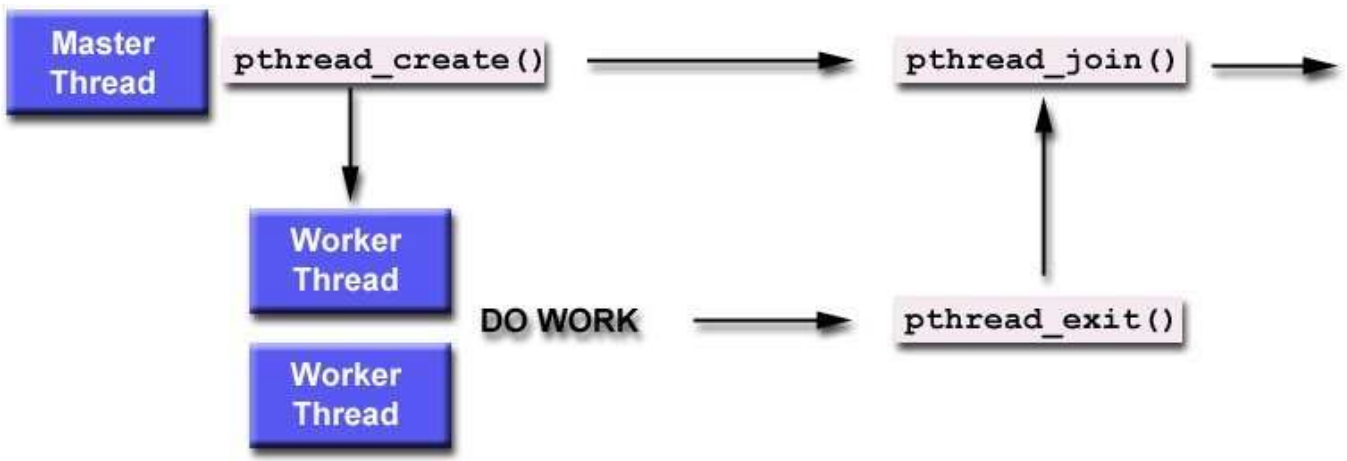
- Jev, kdy alespoň jedno vlákno není schopné vzhledem k paralelnímu souběhu s jiným vláknem pokročit ve výpočtu za danou hranici.

Příklad

- ```
  volatile int a=0;

  P0 {                          P1 {
    while (true) {;               while (a == 0) { };
      a++; a-;                    ...;
    }...
  }                             }
  ```

- Vlákno P1 může na vyznačeném řádku strávit mnoho času.
  ```

Thread-safe a re-entrantní procedury

Thread-safe procedura

- Označení procedury či programu, jejíž kód je bezpečné provádět (vzhledem k sémantice výstupu a stabilitě výpočtu) souběžné několika vlákny bez nutnosti vzájemné domluvy/synchronizace.
- Knihovní funkce nemusí být thread-safe!
  - rand() -> rand_r()

Re-entrantní procedura

- Procedura, jejíž provádění může být v rámci jednoho vlákna přerušeno, kód kompletně vykonán od začátku do konce v rámci téže úlohy, a poté obnoveno/dokončeno přerušené vykonávání kódu.
- Termím pochází z dob, kdy nebyly multitaskingové operační systémy.

Vztah thread-safe a re-entrantních procedur

Neporovnatelné

- Re-entrantní procedura nemusí být thread-safe.
  viz `http://en.wikipedia.org/wiki/Reentrancy_(computing)`
- Thread-safe procedura nemusí být re-entrantní.
  (Problémem je například používání globálních zámků.)

Příklad

- Thread-safe procedura, která není re-entrantní:

```
WC {
  je-li odemčeno, vejdi a zamkni, jinak čekej
  ...
  odemkni a opusť onu místnost
}
```

Thread-safe a re-entrantní procedury

Nebezpečné akce vzhledem k paralelnímu zpracování

- Nekontrolovaný přístup ke globálním proměnným a haldě.
- Uchovávání stavu procedury do globálních proměnných.
- Alokace, dealokace zdrojů globálního rozsahu (soubory, . . .).
- Nepřímý přístup k datům skrze odkazy nebo ukazatele.
- Viditelný vedlejší efekt (modifikace nestálých proměnných).

Bezpečná strategie

- Přístup pouze k lokálním proměnným (zásobník).
- Kód je závislý pouze na argumentech dané funkce.
- Veškeré volané podprocedury a funkce jsou thread-safe.

Procedura, která není thread-safe

```
*myStructure p;

*myStructure function() {
  p=new myStructure;
  return p;
}

P0 {
  *myStructure x;
  x=function();
}
```

```
P1 {
  *myStructure x;
  x=function();
}
```

Přístup ke sdíleným proměnným

Pozorování

- Přístup ke sdíleným proměnným je „kořenem všeho zla“.
- Veškeré modifikace a neatomická čtení globálních proměnných musí být **serializovány**.

Kritická sekce

- Část kódu, jehož provedení je neproložitelné instrukcemi jiného vlákna.
- Realizace kritické sekce musí být odolná vůči plánování.

Zamykání

- Vlákno vstupující do volné kritické sekce, svým vstupem znemožní přístup ostatním vláknům (sekci tzv. zamkne).
- Ostatní vlákna čekají před vstupem do kritické sekce.
- Při odchodu z kritické sekce, je zámek uvolněn, získá ho **náhodně** jedno z čekajících vláken.

Zamykání a související pojmy

Jednoduché řešení zámku

- Sdílená atomicky přistupovaná bitová proměnná, jejíž hodnota indikuje přítomnost procesu/vlákna v přidružené kritické sekci.
- Manipulována při vstupu a výstupu z/do kritické sekce.
- Vyžaduje podporu HW pro atomickou manipulaci.

Aktivní čekání – spinlock

- Dokud neuspěje, vlákno opakovaně zkouší vstoupit do kritické sekce (tj. po tuto dobu neustále provádí tentýž kód).

Uspávání

- Procesy/vlákna se po neúspěchu vstoupit do kritické sekce sami vzdají procesorového kvanta (uspí se).
- Jsou buzeny buď po vypršení časového limitu nebo explicitně jiným běžícím vláknem.

Rizika zamykání

Rizika

- Uváznutí, stárnutí, snížení výkonnosti.

Přístup ke sdíleným globálním proměnným

- Manipulace se zámkem vynucuje vylití cache pamětí.
- Mnoho přístupů k zamykaným proměnným může být úzkým místem výkonu aplikace, z principu nelze odstranit.

Petersonův algoritmus (spinlock, user-space)

- Spravedlivý algoritmus pro řízení vzájemného vyloučení.
- Nezpůsobuje stárnutí ani uváznutí.
- Vyžaduje atomické zápisy do proměnných.
- Citlivý na provádění instrukcí mimo pořadí.

POSIX Thread API

Historie a POSIX standard

Historie

- SMP systémy
- Vlákna implementována jednotlivými výrobci HW
- IEEE POSIX 1003.1c standard

IEEE POSIX 1003.1c

- Programátorský model semaforů a provádění operací v kritické sekci
- Rozhraní pro C
- POSIX threads, PThreads

Jiné normy

- Operační systémy: NT Threads (Win32), Solaris threads, ...
- Programovací jazyky: Java threads, C++11 threading, ...

Základní dělení funkcionality

Správa vláken

- Vytváření, oddělování a spojování vláken
- Funkce na nastavení a zjištění stavu vlákna

Vzájemná vyloučení (mutexes)

- Vytváření, ničení, zamykání a odemykání mutexů
- Funkce na nastavení a zjištění atributů spojených s mutexy

Podmínkové/podmíněné proměnné (conditional variable)

- Slouží pro komunikaci/synchronizaci vláken
- Funkce na vytváření, ničení, "čekání na" a "signalizování při" specifické hodnotě podmínkové proměnné
- Funkce na nastavení a zjištění atributů proměnných

POSIX standard

Přes 60 API funkcí

- `#include <pthread.h>`
- Překlad s volbou `-pthread`

Mnemotechnické předpony funkcí

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

Pracuje se skrytými objekty (Opaque objects)

- Objekty v paměti, o jejichž podobě programátor nic neví.
- Přistupovány výhradně pomocí odkazu (handle).
- Nedostupné objekty a neplatné (dangling) reference.

Atributy objektů

Idea

- Vlastnosti všech vláken, mutexů i podmínkových proměnných nastavovány speciálními objekty.
- Některé vlastnosti entity musí být specifikovány již v době vzniku entity.

Typy atributových objektů

- Vlákna: `pthread_attr_t`
- Mutexy: `pthread_mutexattr_t`
- Podmínkové proměnné: `pthread_condattr_t`

Vznik a destrukce

- Funkce `_init` a `_destroy` s odpovídající předponou
- Parametr odkaz na odpovídající atributový objekt

Správa vláken

Vytváření vlákna

- Každý program má jedno hlavní vlákno
- Další vlákna musí být explicitně vytvořena programem
- Každé vlákno (i vytvořené) může dále vytvářet další vlákna
- Vlákno vytvářeno funkcí `pthread_create`
- Vytvářené vlákno je ihned připraveno k provádění
- Může být plánovačem spuštěno dříve, než se dokončí volání vytvářecí funkce
- Veškerá data potřebná při spuštění vlákna, musí být připravena před voláním vytvářecí funkce
- Maximální počet vláken je závislý na implementaci

Správa vláken

```
int pthread_create (
        pthread_t *thread_handle,
        const pthread_attr_t *attribute,
        void * (*thread_function)(void *),
        void *arg);
```

- `thread_handle` odkaz na vytvořené vlákno
- `attribute` odkaz na atributy vytvořeného vlákna (NULL pro přednastavené nastavení atributů)
- `thread_function` ukazatel na funkci nového vlákna
- `arg` ukazatel na parametry funkce `thread_function`
- Při úspěšném vytvoření vlákna vrací 0

Správa vláken

Ukončení vlákna nastává

- Voláním funkce `pthread_exit`
- Pokud skončí hlavní funkce rodičovského vlákna jinak než voláním `pthread_exit`
- Je-li zrušeno jiným vláknem pomocí `pthread_cancel`
- Rodičovský proces je ukončen (násilně nebo voláním `exit`)

```
void pthread_exit (void *value)
```

- Ukončuje běh vlákna
- Odkazy na prostředky procesu (soubory, IPC, mutexy, ...) otevřené v rámci vlákna se nezavírají
- Data patřící vláknu musí být uvolněna před ukončením vlákna (systém provede uvolnění prostředků až po skončení rodičovského procesu)
- Ukazatel `value` předán při spojení vláken

Správa vláken – příklad

```
1   #include <pthread.h>
2   #include <stdio.h>
3   #define NUM_THREADS 5
4
5   void *PrintHello(void *threadid)
6   {   printf("%d: Hello World!\n", threadid);
7       pthread_exit(NULL);
8   }
9
10  int main (int argc, char *argv[])
11  {   pthread_t threads[NUM_THREADS];
12      for(int t=0; t<NUM_THREADS; t++)
13        pthread_create(&threads[t], NULL,
14                       PrintHello, (void *)t);
15      pthread_exit(NULL);
16  }
```

Správa vláken

```
int pthread_join (pthread_t thread_handle,
                  void **ptr_value);
```

- Čeká na dokončení vlákna `thread_handle`
- Hodnota `ptr_value` je ukazatel na pointer specifikovaný vláknem `thread_handle` při volání `pthread_exit`
- Nutný například pokud main má vracet smysluplnou návratovou hodnotu



Vlastnosti (atributy) vláken

Nespojitelná vlákna (Detached threads)

- Nemohou být spojena voláním funkce `pthread_join`
- Šetří systémové prostředky
- Přednastavené nastavení typu vlákna není vždy zřejmé, proto je doporučeno typ vlákna explicitně nastavit

```
int pthread_detach (
      ptrhead_t *thread_handle)

int pthread_attr_setdetachstate(
      pthread_attr_t *attr,
      int detachstate)

int pthread_attr_getdetachstate(
      pthread_attr_t *attr,
      int *detachstate)
```

Vlastnosti (atributy) vláken

Velikost zásobníku

- Minimální velikost zásobníku není určena
- Při velkém počtu vláken se často stává, že vyhrazené místo pro zásobník je vyčerpáno
- POSIX umožňuje zjistit a nastavit pozici a velikost místa vyhrazeného pro zásobník jednoho vlákna

```
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)
```

Zrušení vlákna

```
int pthread_cancel (
      ptrhead_t *thread_handle)
```

- Žádost o zrušení vlákna `thread_handle`
- Adresované vlákno se může a nemusí ukončit
- Vlákno může ukončit samo sebe
- Při zrušení se provádí úklid dat spojených s rušeným vláknem
- Funkce skončí po odeslání žádosti (je neblokující)
- Návratový kód 0 značí, že adresované vlákno existuje, ne že bylo/bude zrušeno

Vzájemné vyloučení

Motivace

- Vícero vláken provádí následující kód

  ```
  if (my_cost < best_cost) best_cost = my_cost;
  ```

- Nedeterministický výsledek pro 2 vlákna a hodnoty:
  `best_cost==100`, `my_cost@1==50`, `my_cost@2==75`

Řešení

- Umístění kódu do kritické sekce
- `pthread_mutex_t`

Inicializace `mutexu`

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

- Parametr `attribute` specifikuje vlastnosti zámku
- `NULL` znamená výchozí (přednastavené) nastavení

Vzájemné vyloučení

`int pthread_mutex_lock (pthread_mutex_t *mutex_lock)`

- Volání této funkce zamyká `mutex_lock`
- Volání je blokující, dokud se nepodaří mutex zamknout
- Zamknout mutex se podaří pouze jednou jednomu vláknu
- Vlákno, kterému se podaří mutex zamknout je v kritické sekci
- Při opouštění kritické sekce, je vlákno "povinné"mutex odemknout
- Teprve po odemknutí je možné mutex znovu zamknout
- Kód provedený v rámci kritické sekce je po odemčení mutexu globálně viditelný (paměťová bariéra)

`int pthread_mutex_unlock (pthread_mutex_t *mutex_lock)`

- Odemyká `mutex_lock`

Vzájemné vyloučení

Pozorování

- Velké kritické sekce snižují výkon aplikace
- Příliš času se tráví v blokujícím volání funkce `pthread_mutex_lock`

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

- Pokusí se zamknout mutex
- V případě úspěchu vrací 0
- V případě neúspěchu `EBUSY`
- Smysluplné využití komplikuje návrh programu
- Implementace bývá rychlejší než `pthread_mutex_lock` (nemusí se manipulovat s frontami čekajících procesů)
- Má smysl aktivně čekat opakovaným volání `trylock`

Vlastnosti (atributy) mutexů

Normální mutex

- Pouze jedno vlákno může jedenkrát zamknout mutex
- Pokud se vlákno, které má zamčený mutex, pokusí znovu zamknout stejný mutex, dojde k uváznutí

Rekurzivní mutex

- Dovoluje jednomu vláknu zamknout mutex opakovaně
- K mutexu je asociován čítač zamknutí
- Nenulový čítač značí zamknutý mutex
- Pro odemknutí je nutno zavolat `unlock` tolikrát, kolikrát bylo voláno `lock`

Normální mutex s kontrolou chyby

- Chová se jako normální mutex, akorát při pokusu o druhé zamknutí ohlásí chybu
- Pomalejší, typicky používán dočasně po dobu vývoje aplikace, pak nahrazen normálním mutexem

Vlastnosti (atributy) mutexů

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

- Nastavení typu mutexu
- Typ určen hodnotou proměnné `type`
- Hodnota `type` může být
  - `PTHREAD_MUTEX_NORMAL_NP`
  - `PTHREAD_MUTEX_RECURSIVE_NP`
  - `PTHREAD_MUTEX_ERRORCHECK_NP`

IB109 Návrh a implementace paralelních systémů

POSIX Threads – pokračování
Win32 Threads

Jiří Barnat

Základní dělení

Správa vláken

- Vytváření, oddělování a spojování vláken.
- Funkce na nastavení a zjištění stavu vlákna.

Vzájemná vyloučení (mutexes)

- Vytváření, ničení, zamykání a odemykání mutexů.
- Funkce na nastavení a zjištění atributů spojených s mutexy.

Podmínkové/podmíněné proměnné (conditional variable)

- Slouží pro komunikaci/synchronizaci vláken.
- Funkce na vytváření, ničení, "čekání na" a "signalizování při" specifické hodnotě podmínkové proměnné.
- Funkce na nastavení a zjištění atributů proměnných.

Podmínkové proměnné v POSIX Threads

Podmínkové proměnné — Motivace

Motivace I

- Často na jednu kritickou sekci čeká vícero vláken.
- Aktivní čekání – permanentní zátěž CPU.
- Uspávání s timeoutem – netriviální režie, omezená frekvence dotazování se na možnost vstupu do kritické sekce.

Motivace II

- Vlákno realizuje ucelenou, logicky oddělenou funkcionalitu.
- Ta není třeba po celou dobu běhu aplikace.
- Programátorem řízená dočasná deaktivace vlákna.

Podmínkové proměnné

Obecné řešení

- Uspání vlákna, pokud vlákno má/musí čekat.
- Vzbuzení vlákna v okamžiku, kdy je možné pokračovat.

Realizace v POSIX Threads

- Mechanismus označovaný jako **Podmínkové proměnné**.
- Podmínková proměnná vyžaduje použití mutexu.
- Po získání mutexu se vlákno může dočasně vzdát tohoto mutexu a uspat se (v rámci dané podmínkové proměnné).
- Probuzení musí být explicitně signalizováno jiným vláknem.

Podmínkové proměnné

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

- Inicializuje podmínkovou proměnnou.
- Má-li `attr` hodnotu `NULL`, použije se výchozí chování.

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

- Zničí nepoužívanou podmínkovou proměnnou a související datové struktury.

Podmínkové proměnné

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

- Uvolní mutex `mutex_lock` a zablokuje vlákno ve spojení s podmínkovou proměnou `cond`.
- Po návratu vlákno opět vlastní mutex `mutex_lock`.
- Před použitím musí být `mutex_lock` inicializován a zamčen volajícím vláknem.

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

- Signalizuje probuzení jednoho z vláken, uspaných nad podmínkovou proměnou `cond`.

Podmínkové proměnné

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

- Signalizuje probuzení všem vláknům čekajících nad podmínkovou proměnnou `cond`.

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

- Vlákno buď vzbuzeno signálem, nebo vzbuzeno po uplynutí času specifikovaném v `abstime`.
- Při vzbuzení z důvodu uplynutí času, vrací chybu `ETIMEDOUT`, a neimplikuje znovu získání `mutex_lock`.

Podmínkové proměnné – typické použití

```
12    pthread_cond_t is_empty;
13    pthread_mutex_t mutex;


432   pthread_mutex_lock(&mutex);
433   while ( size > 0 )
434     pthread_cond_wait(&is_empty,&mutex);
      ...
456   pthread_mutex_unlock(&mutex);


715   [pthread_mutex_lock(&mutex);]
      ...
721   size=0;
722   pthread_cond_signal(&is_empty);
723   [pthread_mutex_unlock(&mutex);]
```

Další funkce v POSIX Threads

Globální, přesto vláknu specifické proměnné

Problém

- Vzhledem k požadavkům vytváření reentrantních a thread-safe funkcí se programátorům zakazuje používat globální data.
- Případné použití globálních proměnných musí být bezstavové a prováděno v kritické sekci.
- Klade omezení na programátory.

Řešení

- Thread specific data (TSD)
- Globální proměnné, které mohou mít pro každé vlákno jinou hodnotu.

Implementace TSD

Standardní řešení

- Pole indexované jednoznačným identifikátorem vlákna.
- Vlákna musí mít rozumně velké identifikátory.
- Snadný přístup k datům patřící jiným vláknům – potenciální riziko nekorektního kódu.

Řešení POSIX standardu

- Identifikátor (klíč) a asociovaná hodnota.
- Identifikátor je globální, asociovaná hodnota lokální proměnná.
- Klíč – `pthread_key_t`.
- Asociovaná hodnota – univerzální ukazatel, tj. `void *`.

POSIX Klíče

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

- Vytvoří nový klíč (jedná se o globální proměnnou).
- Hodnota asociovaného ukazatele je nastavena na NULL pro všechna vlákna.
- Parametr `destructor` – funkce, která bude nad asociovanou hodnotou vlákna volána v okamžiku ukončení vlákna, pokud bude asociovaná hodnota nenulový ukazatel.
- Parametr `destructor` je nepovinný, lze nahradit NULL.

POSIX Klíče – použití

Zničení klíče a asociovaných ukazatelů

- `int pthread_key_delete (ptrhead_key_t key)`
- Nevolá žádné `destructor` funkce.
- Programátor je zodpovědný za dealokaci objektů před zničením klíče.

Funkce na získání a nastavení hodnoty asociovaného ukazatele

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

Různé

```
ptrhead_t pthread_self ()
```

- Vrací unikátní systémový identifikátor vlákna

```
int pthread_equal (pthread_t thread1,
    pthread_t thread2)
```

- Vrací nenula při identitě vláken `thread1` a `thread2`

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

- První volání této funkce z jakéhokoliv vlákna způsobí provedení kódu `init_routine`. Další volání nemají žádný efekt.

Další funkce v POSIX Threads

Plánování (scheduling) vláken

- Není definováno, většinou je výchozí politika dostačující.
- POSIX Threads poskytují funkce na definici vlastní politiky a priorit vláken.
- Není požadována implementace této části API.

Správa priorit mutexů.

Sdílení podmínkových proměnných mezi procesy.

Vlákna a obsluha POSIX signálů.

Read-Write zámky.

Typické konstrukce

Čtenáři a písaři – WRRM Mapa

Specifikace problému

- Vlákna aplikace často čtou hodnotu, která je relativně méně často modifikována. (Write-Rarely-Read-Many)
- Je žádoucí, aby čtení hodnoty mohlo probíhat souběžně.

Možné problémy

- Souběžný přístup dvou vláken-písařů, může vyústit v nekonzistentní data nebo mít nežádoucí vedlejší efekt, například memory leak.
- Souběžný přístup vlákna-písaře v okamžiku čtení hodnoty jiným vláknem-čtenářem může vyústit v čtení neplatných, nekonzistentních dat.

Čtenáři a písaři – Řešení

Řešení s použitím POSIX Threads

- Čtení a modifikace dat bude probíhat v kritické sekci.
- Přístup do kritické sekce bude řízen pomocí funkcí `pthread_*`.

Další požadavky

- Vlákno-čtenář může vstoupit do kritické sekce, pokud v ní není nebo na ní nečeká žádné vlákno-písař.
- Vlákno-čtenář může vstoupit do kritické sekce, pokud v ní jsou jiná vlákna-čtenáři.
- Přístupy vláken-písařů jsou serializovány a mají přednost před přístupy vláken-čtenářů.

Čtenáři a písaři – Implementace

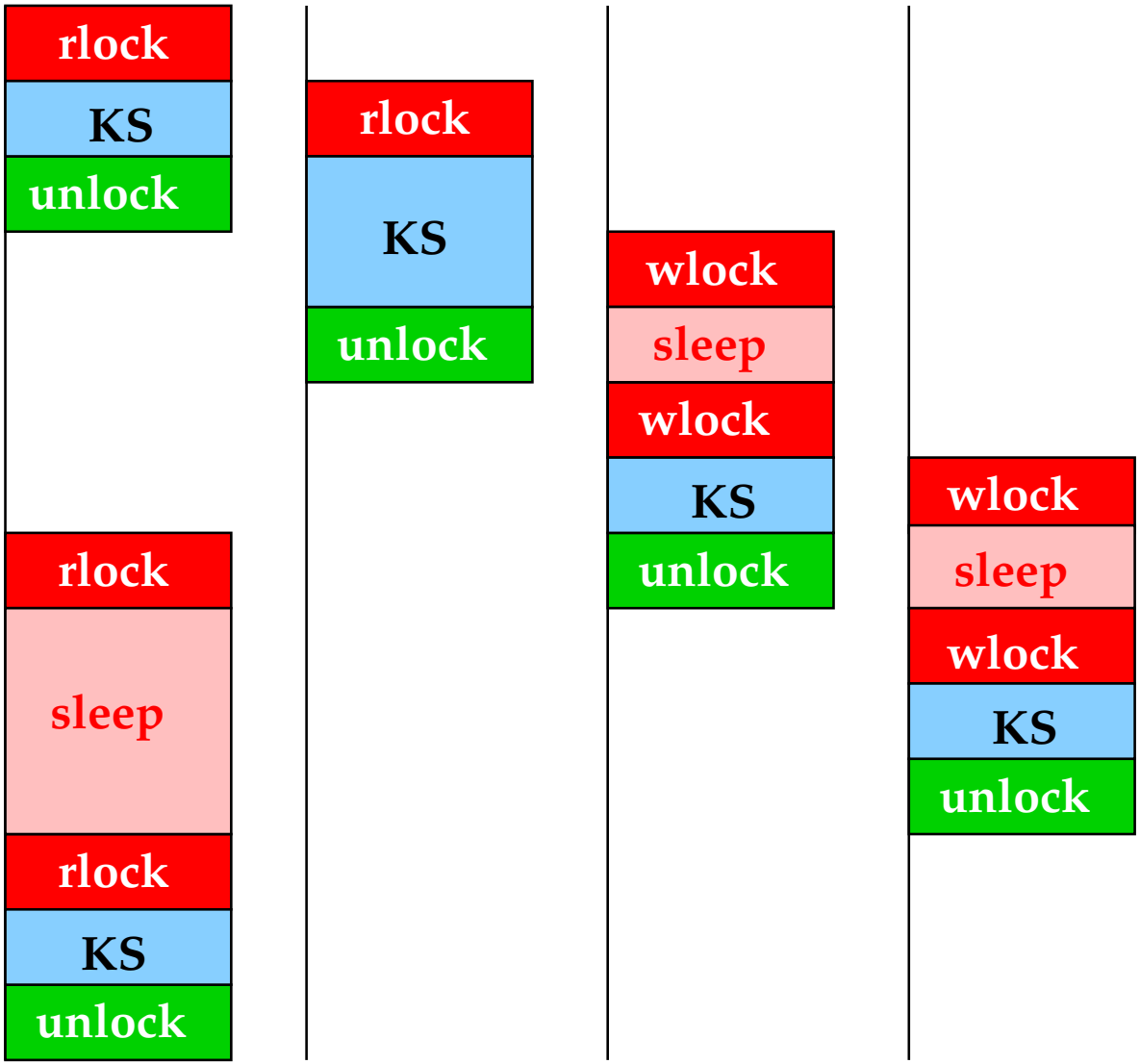
Jednoduché řešení

- Použít jeden `pthread_mutex_t` pro kritickou sekci.
- Vylučuje souběžný přístup vláken-čtenářů.

Lepší řešení

- Implementujeme nový typ zámku – `rwlock_t`
- Funkce pracující s novým zámkem
  - `rwlock_rlock(rwlock_t *l)` – vstup vlákna-čtenáře
  - `rwlock_wlock(rwlock_t *l)` – vstup vlákna-písaře
  - `rwlock_unlock(rwlock_t *l)` – opuštění libovolným vláknem
- Funkce `rwlock` implementovány s využitím podmínkových proměnných z POSIX Thread API.

Čtenáři a písaři – Implementace



Čtenáři a písaři – Implementace

```
1   typedef struct {
2     int readers;
3     int writer;
4     pthread_cond_t readers_proceed;
5     pthread_cond_t writer_proceed;
6     int pending_writers;
7     pthread_mutex_t lock;
8   } rwlock_t;
9
10  void rwlock_init (rwlock_t *l) {
11    l->readers = l->writer = l->pending_writers = 0;
12    pthread_mutex_init(&(l->lock),NULL);
13    pthread_cond_init(&(l->readers_proceed),NULL);
14    pthread_cond_init(&(l->writer_proceed),NULL);
15  }
```

Čtenáři a písaři – Implementace

```
16
17      void rwlock_rlock (rwlock_t *l) {
18        pthread_mutex_lock(&(l->lock));
19        while (l->pending_writers>0 || (l->writer>0)) {
20          pthread_cond_wait(&(l->readers_proceed), &(l->lock));
21        }
22        l->readers++;
23        pthread_mutex_unlock(&(l->lock));
24      }
25
```

Čtenáři a písaři – Implementace

```
26
27      void rwlock_wlock (rwlock_t *l) {
28        pthread_mutex_lock(&(l->lock));
29        while (l->writer>0 || (l->readers>0)) {
30          l->pending_writers++;
31          pthread_cond_wait(&(l->writer_proceed), &(l->lock));
32          l->pending_writers -;
33        }
34        l->writer++;
35        pthread_mutex_unlock(&(l->lock));
36      }
37
```

Čtenáři a písaři – Implementace

```
38
39      void rwlock_unlock (rwlock_t *l) {
40        pthread_mutex_lock(&(l->lock));
41        if (l->writer>0)
42          l->writer=0;
43        else if (l->readers>0)
44          l->readers-;
45        pthread_mutex_unlock(&(l->lock));
46        if ( l->readers == 0 && l->pending_writers >0)
47          pthread_cond_signal( &(l->writer_proceed) );
48        else if (l->readers>0)
49          pthread_cond_broadcast( &(l->readers_proceed) )
50      }
51
```

Čtenáři a písaři – Příklady použití

- Počítání minima

```
2122    ...
2123    rwlock_rlock(&rw_lock);
2124    if (my_min < global_min) {
2125        rwlock_unlock(&rw_lock);
2126        rwlock_wlock(&rw_lock);
2127        if (my_min < global_min) {
2128            global_min = my_min;
2129        }
2130    }
2131    rwlock_unlock(&rw_lock);
2132    ...
```

- Hašovací tabulky
- ...

Bariéry

Specifikace problému

- Synchronizační primitivum
- Vláknu je dovoleno pokračovat po *bariéře* až když ostatní vlákna dosáhly bariéry.
- Naivní implementace přes mutexy vyžaduje aktivní čekání (nemusí být vždy efektivní).

Lepší řešení

- Implementace bariéry s použitím podmínkové proměnné a počítadla.
- Každé vlákno, které dosáhlo bariéry zvýší počítadlo.
- Pokud není dosaženo počtu vláken, podmíněné čekání.

Bariéry – Implementace

```
1   typedef struct {
2     pthread_mutex_t count_lock;
3     pthread_cond_t ok_to_proceed;
4     int count;
5   } barrier_t;
6
7   void barrier_init (barrier_t *b) {
8     b->count = 0;
9     pthread_mutex_init(&(b->count_lock),NULL);
10    pthread_cond_init(&(b->ok_to_proceed),NULL);
11  }
```

Bariéry – Implementace

```
12  void barrier (barrier_t *b, int nr_threads) {
13    pthread_mutex_lock(&(b->count_lock));
14    b->count ++;
15    if (b->count == nr_threads) {
16      b->count = 0;
17      pthread_cond_broadcast(&(b->ok_to_proceed));
18    }
19    else
20      while (pthread_cond_wait(&(b->ok_to_proceed),
21        &(b->count_lock)) != 0);
22    pthread_mutex_unlock(&(b->count_lock));
23  }
```

Bariéry – Efektivita implementace

Problém

- Po dosažení bariéry všemi vlákny, je mutex `count_lock` postupně zamčen pro všechny vlákna
- Dolní odhad na dobu běhu bariéry je tedy $O(n)$, kde n je počet vláken participujících na bariéře

Možné řešení

- Implementace bariéry metodou binárního půlení
- Teoretický dolní odhad na bariéru je $O(n/p + log\ p)$, kde p je počet procesorů

Cvičení

- Implementujte bariéru využívající binárního půlení
- Měřte dopad počtu participujících vláken na dobu trvání lineární a logaritmické bariéry na vámi zvoleném paralelním systému

Chyby, krom nezamykaného přístupu ke globální proměnné

Typické chyby – situace 1

- Vlákno V1 vytváří vlákno V2
- V2 požaduje data od V1
- V1 plní data až po vytvoření V2
- V2 použije neinicializovaná data

Typické chyby – situace 2

- Vlákno V1 vytváří vlákno V2
- V1 předá V2 pointer na lokální data V1
- V2 přistupuje k datům asynchronně
- V2 použije data, která už neexistují (V1 skončilo)

Typické chyby – situace 3

- V1 má vyšší prioritu než V2, čtou stejná data
- Není garantováno, že V1 přistupuje k datům před V2
- Pokud V2 má destruktivní čtení, V1 použije neplatné data

Ladění programů s POSIX vlákny

Valgrind

- Simulace běhu programu.
- Analýza jednoho běhu programu.

Nástroje valgrindu

- Memcheck – detekce nekonzistentního použití paměti.
- Callgrind – jednoduchý profiler.
  - kcachegrind – vizualizace.
- Helgrind – detekce nezamykaných přístupů ke sdíleným proměnným v POSIX Thread programech.

Rozšíření POSIX Threads – nepovinná dle standardu

Barriéry

- `pthread_barrier_t`
- `pthread_barrierattr_t`
- `_init(...)`, `_destroy(...)`, `_wait(...)`

Read-Write zámky

- `pthread_rwlock_t`
- `pthread_rwlockattr_t`
- `_init(...)`, `_destroy(...)`
- `_rdlock(...)`, `_wrlock(...)`,`_unlock(...)`
- `_tryrdlock(...)`, `_trywrlock(...)`
- `_timedrdlock(...)`, `_timedwrlock(...)`

Další způsoby synchronizace

Synchronizace procesů

Problém – jak synchronizovat procesy

- Mutexy z POSIX Threads dle standardu slouží pouze pro synchronizaci vláken v rámci procesu.
- Pro realizaci kritických sekcí v různých procesech je třeba jiných synchronizačních primitiv.
- Podpora ze strany operačního systému.

Semafory

- Čítače používané ke kontrole přístupů ke sdíleným zdrojům.
- POSIX semafory (v rámci procesu)
- System V semafory (mezi procesy)
- Lze využít i pro synchronizaci vláken.

Semafory

Semafor

- Celočíselný nezáporný čítač jehož hodnota indikuje "obsazenost" sdíleného zdroje.
  - Nula – zdroj je využíván a není k dispozici.
  - Nenula – zdroj není využíván, je k dispozici.
- sem_init() – inicializuje čítač zadanou výchozí hodnotou
- sem_wait() – sníží čítač, pokud může, a skončí, jinak blokuje
- sem_post() – zvýší čítač o 1, případně vzbudí čekající vlákno

Semafory vs. mutexy

- Mutex smí odemknout pouze to vlákno, kterého jej zamklo.
- Semafor může být spravován / manipulován různými vlákny.

Monitory

Monitor

- Synchronizační primitivum vyššího programovacího jazyka.
- Označení kódu, který může být souběžně prováděn nejvýše jedním vláknem.
- JAVA – klíčové slovo `synchronized`

Semafory, mutexy a monitory

- Se semafory a mutexy je explicitně manipulováno programátorem.
- Vzájemné vyloučení realizované monitorem je implicitní, tj. explicitní zamykání skrze explicitní primitiva doplní překladač.

Vlákna v MS Windows

Vlákna v MS Windows

Vyšší programovací jazyk

- C++11
- JAVA
- ...

POSIX Thread pro Windows

- Existuje knihovna poskytující POSIX Thread interface.

Nativní rozhraní MS Windows

- Přímá systémová volání (součást jádra OS).
- Pouze rámcově podobná funkcionalita jako POSIX Threads.
- Na rozdíl od POSIX Threads nemá nepovinné části (tudíž neexistují různé implementace téhož).

Windows vs. POSIX Threads – Funkce

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

Win32 vs. POSIX Threads – Synchronizace

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

Win32 vs. POSIX Threads – Základní mapování

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

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

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

Win32 vs. Linux/UNIX – Mapování synchronizace

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

Win32 vs. UNIX – Pozorování

Pozice vláken ve MS Windows

- Silnější postavení než vlákna v Linuxu.
- Synchronizační prostředky fungují i mezi procesy.
- Vlákna ve vlákně (Processes-Threads-Fibers)
- *User-mode scheduling (UMS)* – kooperativní plánování.

Výhody jednoho typu

- Jednou funkcí lze čekat na nekonkrétní vlákno.
- Jednou funkcí lze čekat na vlákno a na mutex.

IB109 Návrh a implementace paralelních systémů

Implementace Lock-Free datových struktur

Jiří Barnat

Motivace

Klasická škola vícevláknového programování

- Přístup ke sdíleným datům musí být chráněný.
- Přístupy k datům se musí serializovat s využitím různých synchronizačních primitiv (mutexy, semafory, monitory).
- Vlákna operují s daty tak, aby se tyto operace jevily ostatním vláknům jako atomické operace.

Problémy

- Prodlevy při přístupu ke sdíleným datům.
- Uváznutí, živost, férovost.
- Korektnost implementace.
- Atomicita operací. (Je `++i` atomické?)

Lock-Free

Lock-free programování

- Programování paralelních (vícevláknových) aplikací bez použití zamykání nebo jiných makro-synchronizačních mechanismů.

Vlastnosti lock-free programování

- Používá se (typicky) jedna jediná atomická konstrukce/instrukce
- Minimální prodlevy související s přístupem k datům
- Neexistuje uváznutí, je garantována živost
- Algoritmicky obtížnější uvažování
- Korektnost algoritmu náchylná na optimalizace překladače

Pojmy

Wait-free procedura

- Procedura, která bez ohledu na souběh dvou a více vláken dokončí svou činnost v konečném čase, tj. neexistuje souběh, který by nutil proceduru nekonečně dlouho čekat, či provádět nekonečně mnoho operací.

Lock-free procedura

- Procedura, která garantuje, že při libovolném souběhu mnoha soupeřících vláken, vždy alespoň jedno vlákno úspěšně dokončí svou činnost. Některá soupeřící vlákna mohou být libovolně dlouho nucena odkládat dokončení své činnosti.

Historie

Maurice Herlihy

- Článek: *Wait-Free Synchronization* (1991)
- Ukázal, že konstrukce jako
  - *test-and-set*
  - *swap*
  - *fetch-and-add*
  - *fronty s atomickými operacemi vložení a výběru*

  nejsou vhodné pro budování lock-free datových struktur pro vícevláknové aplikace.
- Ukázal, že existují konstrukce, které vhodné jsou (např. CAS).
- Dijkstrova cena za distribuované počítání (2003)

  `http://www.podc.org/dijkstra/2003.html`

Důsledek

- Současné procesory mají odpovídající HW podporu pro CAS.

Konstrukce *compare-and-swap* (CAS)

Sémantika daná pseudo-kódem:

```
template <class T>
bool CAS(T* addr, T exp, T val) {
    if (*addr == exp) {
        *addr = val;
        return true;
    }
    return false;
}
```

Slovní popis

- CAS porovná obsah specifikované paměťové adresy addr s očekávanou hodnotou exp a v případě rovnosti nahradí obsah paměťové adresy novou hodnotou val. O úspěchu či neúspěchu informuje uživatele návratovou hodnotou. Celá procedura proběhne atomicky.

Princip použití instrukce CAS

Postup při přístupu ke sdíleným datům

- Přečtu stávající hodnotu sdíleného objektu
- Připravím novou hodnotu sdíleného objektu
- Aplikuji instrukci CAS

Návratová hodnota

- *True* – Objekt nebyl v mezičase modifikován, nově vypočítaná hodnota je platná a je uložena ve sdíleném objektu.
- *False* – Objekt byl v mezičase modifikován (z jiného vlákna), instrukce CAS neměla žádný efekt a je nutné celý postup opakovat.

Nebezpečí použití CAS – ABA problém

Klíčová vlastnost

- Modifikace objektu proběhnuvší mezi načtením hodnoty objektu a aplikací instrukce CAS nesmí vyprodukovat tutéž hodnotu sdíleného objektu.

Možný chybový scénář

- Hodnota objektu, načtená vláknem A za účelem použití v následné instrukci CAS, je `x`.
- Před použitím instrukce CAS vláknem A, je objekt modifikována jinými vlákny, tj. nabývá hodnot různých od `x`.
- V okamžiku aplikace instrukce CAS vláknem A, má objekt opět hodnotu `x`.
- Vlákno A nepozná, že se hodnota objektu měnila.
- Následná aplikace instrukce CAS uspěje.

CAS – provádění instrukcí mimo pořadí a cena

Provádění instrukcí mimo pořadí

- Pokud používáme CAS na zpřístupnění nějakých dat, je potřeba zajistit, aby předcházející inicializace proměnných byly již v okamžiku vykonání instrukce CAS vykonány.
- Vyžaduje použití paměťové bariéry.
- Dotčené proměnné musejí být označeny jako nestálé.

Cena

- Použití CAS odstranilo režii související se zamykáním.
- Zůstává však režie související s koherencí cache pamětí.

Programování s CAS

Win32

- InterlockedCompareExchange(...)

Asembler i386, (pro x86_64 nutné přejmenovat edx na rdx)

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

GCC – zabudované funkce

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

WRRM Mapa – Příklad

WRRM Mapa

Write Rarely Read Many Mapa

- Zprostředkovává překlad jedné entity na jinou. (Klíč→Hodnota).
- Příklad – kurz Koruny vzhledem k jiným měnám
  - Mění se jednou denně.
  - Používá se při každé transakci.

Možné implementace s využitím STL

- map, hash_map
- assoc_vector (uspořádané dvojice)

Použití

- Map <Key, Value > mojeMapa;

Implementace s použitím Mutexů

```
template <class K, class V>
class WRRMMap {
    Mutex mtx_;
    Map <K,V> map_;
public:

    V Lookup(constK& k) {
        Lock lock(mtx_);
        return map_[k];
    }

    void Update(const K& k, const V& v) {
        Lock lock(mtx_);
        map_.insert(make_pair(k,v));
    }

};
```

Implementace s použitím instrukce CAS

Operace čtení

- Probíhá zcela bez zamykání.

Operace zápisu

- Vytvoření kopie stávající mapy.
- Modifikace/přidání dvojice do vytvořené kopie.
- Atomická záměna nové verze mapy za předcházející.

Reálné omezení CAS

- Obecné použití schématu CAS na WRRMMap by vyžadovalo atomickou změnu relativně rozsáhlé oblasti paměti.
- HW podpora pro CAS je omezena na několik bytů (typicky jedno, nebo dvě slova procesoru).
- Atomickou záměnu provedeme přes ukazatel.

Implementace s použitím instrukce CAS

```
template <class K, class V>
class WRRMMap {
    Map <K,V>* pMap_;
public:
    V Lookup(constK& k) {
        return (*pMap_) [k];
    }
    void Update(const K& k, const V& v) {
    Map <K,V>* pNew=0
    do {
        Map <K,V>* pOld = pMap_;
        delete pNew;                          //if (pNew==0) nothing happens
        pNew = new Map<K,V>(*pOld);
        (*pNew)[k] = v;
    } while (!CAS(&pMap_, pOld, pNew));
    // DON'T delete pOld;
    }
};
```

WRRM Map – vlastnosti a problém dealokace

Proč je nutná instrukce CAS a nestačí jen `pOld = pNew`?

- Vlákno A udělá kopii mapy.
- Vlákno B udělá kopii mapy, vloží nový klíč a dokončí operaci.
- Vlákno A vloží nový klíč.
- Vlákno A nahradí ukazatel, vše, co vložilo B, je ztraceno.

Update

- Je lock-free, ale není wait-free.

Správa paměti

- Update nemůže uvolnit starou kopii datové struktury, jiné vlákno může nad datovou strukturou provádět operaci čtení.
- Možné řešení: Garbage collector (JAVA)

WRRM Map nefunkční řešení dealokace – odložený delete

Odložená dealokace paměti

- Místo delete, se spustí (asynchronně) nové vlákno.
- Nové vlákno počká 200ms a pak provede dealokaci.

Myšlenka

- Nové operace probíhají nad novou kopií, za 200ms se všechny započaté operace nad starou kopií dokončí a bude bezpečné strukturu dealokovat.

Problémy

- Krátkodobé intenzivní přepisování hodnot nebo vkládání nových hodnot může způsobit netriviální paměťové nároky.
- Není garantováno, že se veškeré operace čtení z jiných vláken za daný časový limit dokončí.

WRRM Map a počítání odkazů

Nápad

- Napodobíme metodu používanou při automatickém uvolňování paměti k tomu, abychom mohli explicitně dealokovat strukturu.
- Počítání odkazů – s každým ukazatelem je svázáno číslo, které udává počet vláken, jež tento ukazatel ještě používají.

Modifikace WRRM mapy

- *Procedura Update* provádí podmíněnou dealokaci, tj. dealokuje objekt odkazovaný ukazatelem, pouze pokud žádné jiné vlákno ukazatel nepoužívá.
- *Procedura Lookup* postupuje tak, že zvýší čítač spojený s ukazatelem, přistoupí ke struktuře skrze tento ukazatel, sníží čítač po ukončení práce se strukturou a podmíněně dealokuje strukturu.

WRRM Map a počítání odkazů – nefunkční řešení

Čítač asociovaný s ukazatelem MAP<K,V>*

- **template** <**class** K, **class** V>
  class WRRMMap {
  typedef std::pair<Map<K,V>*,**unsigned**> Data;
  Data* pData_;
  ...
  }
- CAS instrukce nad ukazatelem pData_
- Podmíněná dealokace:
  `if (pData_->second==0) delete (pData_->first);`

Problém v proceduře Lookup

- Vlákno A načte strukturu Data (přes *pDate_) a je přerušeno.
- Vlákno B vloží klíč, sníží čítač a dealokuje `*pOld->first`.
- Vlákno A zvýší čítač, ale přistoupí k neplatnému ukazateli.

WRRM Map a počítaní odkazů – CAS2

Problém předchozí verze

- Akce uchopení ukazatele a zvýšení odpovídajícího čítače nebyly atomické.

Řešení

- Pomocí jedné instrukce CAS je třeba přepnout ukazatel a korektně manipulovat s čítačem.
- Teoreticky je možné implementovat CAS pracující s více strukturami zároveň, ovšem ztrácí se efektivita, pokud neexistuje odpovídající HW podpora.
- Moderní procesory mají podporu pro instrukci CAS pracující se dvěma po sobě uloženými slovy procesoru (CAS2).

WRRM Map s využitím CAS2

Myšlenka

- **template** <**class** K, **class** V>
  class WRRMMap {
  typedef std::pair<Map<K,V>*,**unsigned**> Data;
  Data data_;
  ...
  }
- Struktura Data je tvořena dvěma slovy: ukazatel a čítač
- Ukazatel a čítač jsou uloženy v paměti vedle sebe.
- Strukturu je možné modifikovat pomocí instrukce CAS2.

WRRM Map s využitím CAS2 – Lookup

```
V Lookup (const K& k) {
    Data old;
    Data fresh;
    do {
        old = data_;
        fresh = old;
        ++fresh.second;
    } while (!CAS2(&data_, old, fresh));
    V temp = ((*fresh.first)[k]
    do {
        old = data_;
        fresh = old;
        --fresh.second;
    } while (!CAS2(&data_, old, fresh));
    if (fresh.second == 0) { delete fresh.first; }
    return temp;
}
```

WRRM Map s využitím CAS2 – stále nefunkční

Otázka

- Umíme atomicky realizovat počítání odkazů, je tedy navrhované řešení korektní?

Problém

- Zvýšení a snížení čítače procedurou Lookup je ve zcela nezávislých blocích. Pokud se mezi provedením těchto bloků realizuje nějaká procedura Update, tak přičtení a odečtení jedničky k čítači proběhne nad jinými ukazateli.
- Riziko předčasné dealokace.
- Ztráta ukazatelů na staré kopie – memory leak.

Řešení

- Čítač spojený s ukazatelem použijeme jako stráž.
- Procedura Update bude provádět změny struktury jen tehdy, pokud žádné jiné vlákno ke struktuře nepřistupuje.

WRRM Map s využitím CAS2 – realizace Update

Odkládání provedení modifikace v proceduře `Update`

- Atomické nahrazení ukazatele se děje v okamžiku, kdy jsou všechna ostatní vlákna mimo proceduru `Lookup`.
- Časové intervaly, po které se jednotlivá vlákna nacházejí v proceduře `Lookup` čtenářům se však mohou překrývat.
- Čítač po celou dobu existence jiného vlákna v proceduře `Lookup` neklesá na minimální hodnotu a procedura `Update` tzv. hladoví (starve).

Optimalizace procedury `Update`

- Při opakovaných neúspěších instrukce CAS dochází k opakovanému kopírování původní struktury a následnému mazání vytvořené kopie.
- Neefektivní opakované kopírování lze odstranit pomocí pomocného ukazatele `last`.

WRRM Map s využitím CAS2 – Update

```
void Update (const K& k, const V& v) {
    Data old;
    Data fresh;
    old.second = 1;
    fresh.first = 0;
    fresh.second = 1;
    Map<K,V>* last = 0;
    do {
        old.first = data_.first;
        if (last != old.first) {
            delete fresh.first;
            fresh.first = new Map<K,V>(old.first);
            fresh.first->insert(make_pair(k,v));
            last = old.first;
        }
    } while (!CAS2(&data_, old, fresh));
    delete old.first;
}
```

WRRM Map – pozorování ohledně realizace s CAS2

Lookup

- Není wait-free, inkrementace a dekrementace čítače interferuje s procedurou Update.
- Volání procedur Update je málo – nevadí.

Update

- Není wait-free, interferuje s procedurou Lookup.
- Volání procedur Lookup je mnoho – problém.

Čeho jsme dosáhli

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

Hazardní ukazatele

aneb tak trochu

"Lock-Free Garbage Collector"

Hazardní ukazatele

Motivace

- Dealokace datových struktur v kontextu lock-free programování je obtížná.
- Ukazatel na datový objekt nerozliší, zda je možné, objekt uvolnit z paměti, či nikoliv.
- Čítače použití ukazatelů nejsou dobré řešení.

Princip řešení pomocí hazardních ukazatelů

- Vlákna vystavují ostatním vláknům seznam ukazatelů, které momentálně používají – tzv. **hazardní ukazatele**.
- Bezpečně lze dealokovat pouze objekty, které nejsou odkazovány hazardními ukazateli.

WRRM Mapa a hazardní ukazatele

Původní problém lock-free implementace WRRM Mapy

- Procedura update vytvoří kopii mapy, modifikuje ji, nahradí touto kopií aktuální mapu a starou mapu dealokuje.
- Dealokace staré mapy může interferovat s probíhající procedurou Lookup jiného vlákna.

Řešení

- WRRM Mapa udržuje seznam ukazatelů, které jsou momentálně používány nějakým vláknem v proceduře *Lookup*.
- *Lookup* – vkládá a odebírá ukazatel do seznamu.
- *Update* – uchovává (per-thread) již neplatné ukazatele a příležitostně je prochází a dealokuje ty, které nejsou hazardní.
- Hazardní ukazatele jsou uchovávány ve sdílené datové struktuře $\Longrightarrow$ je třeba ošetřit paralelní přístupy.

Schrána pro hazardní ukazatel – Třída HPRecType

Spojový seznam

- Záznam seznamu obsahuje ukazatel a příznak validity:

  `int` active_

  `void*` pHazard_

Metoda Acquire()

- Vytvoří nebo znovu použije neplatný objekt seznamu a vrátí volajícímu ukazatel na tento objekt.
- Použije se pro zveřejnění používaného ukazatele.

Metoda Release()

- Použije se pro zneplatnění objektu, tj. oznámení, že ukazatel uložený v tomto objektu již není dále používán.

Schrána pro používané ukazatele – Třída HPRecType

```
class HPRecType {
    HPRecType * pNext_;
    int active_;
    static HPRectType* pHead_;
    static int listLen_;
public:
    void * pHazard_;
    static HPRecType* Head() { return pHead_; }
    static HPRecType* Acquire() {
    ...
    }
    static void Release(HPRecType* p) {
    p->pHazard_ = 0;                    // Order matters, pHazard_=0 first
    p->active_ = 0;
    }
}
```

Objekt pro používané ukazatele

```
static HPRecType* Acquire() {
    HPRecType *p = pHead_;
    for(; p; p=p->pNext_) {                                  // Try to reuse
        if (p->active_ or !CAS(&p->active_,0,1)) continue;
        return p;
    }
    int oldLen;                                             // Increment the length
    do {
        oldLen = listLen_;
    } while (!CAS(&listLen_,oldLen, oldLen+1));
    HPRecType *p = new HPRecType;                            // Allocate new slot
    p->active_ = 1;
    p->pHazard_ = 0;
    do {                                                    // Push it to the front
        old = pHead_;
        p->pNext_ = old;
    } while (!CAS(&pHead_, old , p));
    return p;
}
```

Seznam odložených ukazatelů určených k dealokaci

Princip

- Ukazatele na instance určené k dealokaci jsou schromažďovány do seznamu odložených ukazatelů.
- Každé vlákno má svůj vlastní seznam.

Retire()

- Nahrazuje funkci **delete**, odkládá ukazatel do seznamu.
- Je-li seznam příliš dlouhý, volá proceduru Scan, která ze seznamu odstraní nadále nepoužívané ukazatele.
- Příliš dlouhý – dáno parametrem R.

Scan()

- Vytvoří kopii seznamu používaných ukazatelů a seznam setřídí.
- Pro každý odložený ukazatel hledá binárním půlením v seznamu používaných ukazatelů, zda je ještě používán.
- Nadále nepoužívané objekty dealokuje.

Seznam odložených ukazatelů určených k dealokaci

```
class HPRecType {
    ...
};

___per_thread___ vector<Map<K,V>*> rlist;

template <class K, class V>
class WRRMMap {
    ...
private:
    static void Retire(Map<K,V>* pOld) {
        rlist.push_back(pOld);
        if (rlist.size() >= R)
            Scan(HPRecType::Head());
    }
    void Scan(HPRecType* head) {
        ...
    }
};
```

Dealokace odložených ukazatelů

```
void Scan(HPRecType* head) {
    vector<void*> hp;                  // collect non-null hazard pointers
    while (head) {
        void* p = head->pHazard_;
        if (p) hp.push_back(p);
        head = head->pNext_;
    }
    sort(hp.begin(),hp.end(), less<void*>());
    vector<Map<K,V>*>::iterator i = rlist.begin();
    while (i!=rlist.end()) {                    // for every retired pointer
        if (!binary_search(hp.begin(),hp.end(),*i) {    // if not used anymore
            delete *i;                                          // delete it
            if (&*i != &rlist.back())                          //and dequeue it
                *i = rlist.back();              // replace it with the last one
            rlist.pop_back();                           // dequeue the last one

        } else {
            ++i;
        }
    }
};
```

Modifikace základních procedur WRRM Mapy – Update

```
void Update(const K& k, const V& v) {
    Map <K,V>* pNew=0
    do {
        Map <K,V>* pOld = pMap_;
        delete pNew;                          //if (pNew==0) nothing happens
        pNew = new Map<K,V>(*pOld);
        (*pNew)[k] = v;
    } while (!CAS(&pMap_, pOld, pNew));
    Retire(pOld);
}
```

Modifikace základních procedur WRRM Mapy – Lookup

```
V Lookup(const K& k) {
    HPRecType *pRec = HPRecType::Acquire();
    Map<K,V> *ptr;
    do {
        ptr = pMap_;
        pRec -> pHazard_ = ptr;
    } while (pMap_ != ptr);          // is ptr still valid? if so, go on
    V result (*ptr) [k];
    HPRecType::Release(pRec);
    return result;
}
```

WRRM Mapa – Máme hotovo

WRRM Mapa a Hazardní ukazatele

- Volání procedury Update interferuje s procedurou Lookup.
- Procedura Lookup není wait-free.
- Předpokládáme přístup v režimu Write Rarely, takže to nevadí.

Hazardní ukazatele

- Možné řešení problému deterministické dealokace v případě, že systém nepodporuje garbage collection.
- Obecně je možné udržovat vícero hazardních ukazatelů na jedno vlákno.
- Amortizovaná složitost je konstantní.

Návrh Lock-Free datových struktur

- Je možné navrhnout lock-free datové struktury.
- Zajímavá algoritmika.
- Obtížné, pokud chceme deterministické uvolňování paměti.
- Vhodné pro prostředí s Garbage Collectorem (JAVA).

Další programátorská rozhraní

Alternativní API pro lock-free programování

MCAS

- Rozšíření standardní instrukce CAS pro použití s libovolně velkou datovou strukturou.

Transakční paměť

- Paměť je modifikována v jednotlivých transakcích.
- Transakce seskupuje mnoho čtení a zápisů do paměti – je schopna obsáhnout komplexní modifikaci datových struktur.
- Základním manipulovatelným objektem je slovo procesoru, tj. obsah jedné paměťové buňky.
- Příklad: přesun prvku v dynamicky zřetězeném seznamu.

LL/SC

Load-Link/Store-Conditional

- Dvojice instrukcí, která dohromady realizuje CAS.
- LL načte hodnotu a SC ji přepíše, pokud nebyla modifikována. Za modifikaci se považuje i přepsání na tutéž hodnotu.
- LL/SC stejná síla jako CAS, avšak nemá ABA problém.
- HW podpora: Alpha, PowerPC, MIPS, ARM

Problémy

- Změna kontextu se v praxi považuje za modifikaci místa.
- Teoreticky není možné realizovat wait-free proceduru.
- Obtížné ladění.

IB109 Návrh a implementace paralelních systémů

Pokročilá rozhraní pro implementaci paralelních aplikací

Jiří Barnat

Jiný způsob programování v prostředí se sdílenou pamětí

Nevýhody POSIX Threads a Lock-free přístupu

- Na příliš nízké úrovni
- Vhodné pro systémové programátory
- „Příliš složitý přístup na řešení jednoduchých věcí.“

Co bychom chtěli

- Paralelní konstrukce na úrovni programovacího jazyka
- Prostředek vhodný pro aplikační programátory
- Snadné vyjádření běžně používaných paralelních konstrukcí

OpenMP

Myšlenka OpenMP

Myšlenka

- Programátor specifikuje co chce, ne jak se to má udělat.
- Náznak deklarativního přístupu v imperativním programování.

Realizace

- Programátor informuje překladač o zamýšlené paralelizaci uvedením **značek ve zdrojovém kódu** a označením bloků.
- Při překladu překladač sám doplní nízkoúrovňovou realizaci paralelizace.

Styl programování s OpenMP

OpenMP nabízí

- Pragma direktivy překladače

  `#pragma omp direktiva [seznam klauzulí]`
- Knihovní funkce
- Proměnné prostředí

Překlad kódu

- Překladač podporující standard OpenMP
  - při překladu pomocí GCC je nutná volba `-fopenmp`
  - `g++ -fopenmp myapp.c`
- Podporováno nejpoužívanějšími překladači (i Visual C++)
- Možno přeložit do sekvenčního kódu

WWW

- `http://www.openmp.org`

Direktiva parallel – příklad v C++

```
1  #include <omp.h>
2  main ()
3  {
4    int nthreads, tid;
5    #pragma omp parallel private(tid)
6    {
7      tid = omp_get_thread_num();
8      printf("Hello World from thread = %d\n", tid);
9      if (tid == 0)
10     {
11       nthreads = omp_get_num_threads();
12       printf("Number of threads = %d\n", nthreads);
13     }
14   }
15 }
```

Direktiva `parallel`

Použití

- Strukturovaný blok, tj. `{...}`, následující za touto direktivou se provede paralelně.
- Mimo paralelní bloky se kód vykonává sekvenčně.
- Vlákno, které narazí na tuto direktivu se stává hlavním vláknem (master) a má identifikaci vlákna rovnou 0.

Podmíněné spuštění

- Klauzule: `if (výraz typu bool)`
- Vyhodnotí-li se výraz na `false` direktiva `parallel` se ignoruje a následující blok je proveden pouze v jedné kopii.

Stupeň paralelismu

- Počet vláken.
- Přednastavený počet specifikován proměnnou prostředí.
- Klauzule: `num_threads (výraz typu int)`

Direktiva `parallel` – datová lokalita

Klauzule: `private (seznam proměnných)`

- Vyjmenované proměnné se zduplikují a stanou se lokální proměnné v každém vlákně.

Klauzule: `firstprivate (seznam proměnných)`

- Viz `private` s tím, že všechny kopie proměnných jsou inicializované hodnotou originální kopie.

Klauzule: `shared (seznam proměnných)`

- Vyjmenované proměnné budou explicitně existovat pouze v jedné kopii.
- Přístup ke sdíleným proměnným nutno serializovat.

Klauzule: `default ([shared|none])`

- `shared`: všechny proměnné jsou sdílené, pokud není uvedeno jinak.
- `none`: vynucuje explicitní uvedení každé proměnné v klauzuli `private` nebo v klauzuli `shared`.

Direktiva `parallel` – redukce

Klauzule: `reduction (operátor: seznam proměnných)`

- Při ukončení paralelního bloku jsou vyjmenované privátní proměnné zkombinovaný pomocí uvedeného operátoru.
- Kopie uvedených proměnných, které jsou platné po ukončení paralelního bloku, jsou naplněny výslednou hodnotou.
- Proměnné musejí být skalárního typu (nesmí být pole, struktury, atp.).
- Použitelné operátory: +, *, -, &, |, ^ , && a ||

Direktiva `for`

Použití

- Iterace následujícího for-cyklu budou provedeny paralelně
- Musí být použito v rámci bloku za direktivou `parallel` (jinak proběhne sekvenčně).
- Možný zkrácený zápis: `#pragma omp parallel for`

Klauzule: `private`, `firstprivate`, `reduction`

- Stejné jako pro direktivu `parallel`.

Klauzule: `lastprivate`

- Hodnota privátní proměnné ve vláknu zpracovávající poslední iteraci for cyklu je uložena do kopie proměnné platné po skončení cyklu.

Direktiva `for`

Klauzule: `ordered`

- Bloky označené direktivou `ordered` v těle paralelně prováděného cyklu jsou provedeny v tom pořadí, v jakém by byly provedeny sekvenčním programem.
- Klauzule `ordered` je povinná, pokud tělo cyklu obsahuje `ordered` bloky.

Klauzule: `nowait`

- Jednotlivá vlákna se nesynchronizují po provedení cyklu.

Klauzule: `schedule (typ plánování[,velikost])`

- Určuje jak budou iterace rozděleny/mapovány mezi vlákna.
- Implicitní plánování je závislé na implementaci.

Direktiva `for` – Plánování iterací

`static`

- Iterace cyklu rozděleny do bloků o specifikované velikosti.
- Bloky staticky namapovány na vlákna (round-robin).
- Pokud není uvedena velikost, iterace rozděleny mezi vlákna rovnoměrně (pokud je to možné).

`dynamic`

- Bloky iterací cyklu v počtu specifikovaném parametrem `velikost` přidělovány vláknům na žádost, tj. v okamžiku, kdy vlákno dokončilo předchozí práci.
- Výchozí velikost bloku je 1.

Direktiva `for` – Plánování iterací

`guided`

- Bloky iterací mají velikost proporcionální k počtu nezpracovaných iterací poděleným počtem vláken.
- Specifikována velikost k, udává minimální velikost bloku (výchozí hodnota 1).
- Příklad:
  - $k = 7$, 200 volných iterací, 8 vláken
  - Velikosti bloků: 200/8=25, 175/8=21, ..., 63/8 = 7, ...

`runtime`

- Typ plánování určen až za běhu proměnnou `OMP_SCHEDULE`.

Direktiva `sections`

Použití

- Strukturované bloky, každý označený direktivou `section`, mohou být v rámci bloku označeným direktivou `sections` provedeny paralelně.
- Možný zkrácený zápis `#pragma omp parallel sections`
- Umožňuje definovat různý kód pro různá vlákna.

Klauzule: `private`, `firstprivate`, `reduction`, `nowait`

- Stejné jako v předchozích případech

Klauzule: `lastprivate`

- Hodnoty privátních proměnných v poslední sekci (dle zápisu kódu) budou platné po skončení bloku `sections`.

Direktiva sections – příklad

```
1  #include <omp.h>
2  main ()
3  {
4    #pragma omp parallel sections
5    {
6      #pragma omp section
7      {
8        printf("Thread A.");
9      }
10     #pragma omp section
11     {
12       printf("Thread B.");
13     }
14   }
15 }
```

Vnořování direktiv `parallel`

Nevnořený paralelismus

- Direktiva `parallel` určuje vznik oblasti paralelního provádění.
- Direktivy `for` a `sections` určují jak bude práce mapována na vlákna vzniklé dle rodičovské direktivy parallel.

Vnořený paralelismus

- Při nutnosti paralelismu v rámci paralelního bloku, je třeba znovu uvést direktivu `parallel`.
- Vnořování je podmíněné nastavením proměnné prostředí `OMP_NESTED` (hodnoty `TRUE`, `FALSE`).
- Typické použití: vnořené for-cykly
- Obecně je vnořování direktiv v OpenMP poměrně komplikované, nad rámec tohoto tutoriálu.

Direktiva `barrier`

Bariéra

- Místo, které je dovoleno překročit, až když k němu dorazí všechna ostatní vlákna.
- Direktiva bez klauzulí, tj. `#pragma omp barrier`.
- Vztahuje se ke strukturálně nejbližší direktivě `parallel`.
- Musí být voláno všemi vlákny v odpovídajícím bloku direktivy `parallel`.

Poznámka ke kódování

- Direktivy překladače nejsou součástí jazyka.
- Je možné, že v rámci překladu bude vyhodnocen blok, ve kterém je umístěna direktiva bariéry, jako neproveditelný blok a odpovídající kód nebude ve výsledném spustitelném souboru vůbec přítomen.
- Direktivu `barrier`, je nutné umístit v bloku, který se bezpodmínečně provede (zodpovědnost programátora).

Direktiva `single` a `master`

Direktiva `single`

- V kontextu paralelně prováděného bloku je následující strukturní blok proveden pouze jedním vláknem, přičemž není určeno kterým.

Klauzule: `private`, `firstprivate`

Klauzule: `nowait`

- Pokud není uvedena, tak na konci strukturního bloku označeného direktivou `single` je provedena bariéra.

Direktiva `master`

- Speciální případ direktivy `single`.
- Tím vláknem, které provede strážený blok, bude hlavní (`master`) vlákno.

Direktiva `critical` a `atomic`

Direktiva `critical`

- Následující strukturovaný blok je chápán jako kritická sekce a může být prováděn maximálně jedním vláknem v daném čase.
- Kritická sekce může být pojmenována, souběžně je možné provádět kód v kritických sekcích s jiným názvem.
- Pokud není uvedeno jinak, použije se implicitní jméno.
- `#pragma omp critical [(name)]`

Direktiva `atomic`

- Nahrazuje kritickou sekci nad jednoduchými modifikacemi (updaty) proměnných v paměti.
- Atomicita se aplikuje na jeden následující výraz.
- Obecně výraz musí být jednoduchý (jeden *load* a *store*).
- Neatomizovatelný výraz: `x = y = 0;`

Direktiva `flush`

Problém (nestálé proměnné)

- Modifikace sdílených proměnných v jednom vlákně může zůstat skryta ostatním vláknům.

Řešení

- Explicitní direktiva pro kopírování hodnoty proměnné z registru do paměti a zpět.
- `#pragma omp flush [(seznam)]`

Použití

- Po zápisu do sdílené proměnné.
- Před čtením obsahu sdílené proměnné.
- Implicitní v místech bariéry a konce bloků (pokud nejsou bloky v režimu `nowait`).

Direktiva `threadprivate` a `copyin`

Problém (thread-private data)

- Při statickém mapování na vlákna je drahé při opakovaném vzniku a zániku vláken vytvářet kopie privátních proměnných.
- Občas chceme privátní globální proměnné.

Řešení

- Perzistentní privátní proměnné (přetrvají zánik vlákna).
- Při znovuvytvoření vlákna, se proměnné znovupoužijí.
- `#pragma omp threadprivate (seznam)`

Omezení

- Nesmí se použít dynamické plánování vláken.
- Počet vláken v paralelních blocích musí být shodný.

Direktiva `copyin`

- Jako `threadprivate`, ale s inicializací.
- Viz `private` versus `firstprivate`.

OpenMP knihovní funkce – Počet vláken

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

- Specifikuje kolik vláken se vytvoří při příštím použití direktivy `parallel`.
- Musí být použito před samotnou konstrukcí `parallel`.
- Je přebito klauzulí `num_threads`, pokud je přítomna.
- Musí být povoleno dynamické modifikování procesů (`OMP_DYNAMIC`, `omp_set_dynamic()`).

```
int omp_get_num_threads ()
```

- Vrací počet vláken v týmu strukturálně nejbližší direktivy `parallel`, pokud neexistuje, vrací 1.

OpenMP knihovní funkce – Počet vláken a procesorů

`int omp_get_max_threads ()`

- Vrací maximální počet vláken v týmu.

`int omp_get_thread_num ()`

- Vrací unikátní identifikátor vlákna v rámci týmu.

`int omp_get_num_procs ()`

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

`int omp_in_parallel ()`

- Vrací nenula pokud je voláno v rozsahu paralelního bloku.

OpenMP knihovní funkce – Kontrola vytváření vláken

```
void omp_set_dynamic (int dynamic_threads)
int omp_get_dynamic()
```

- Nastavuje a vrací, zda je programátorovi umožněno dynamicky měnit počet vláken vytvořených při dosažení direktivy `parallel`.
- Nenulová hodnota `dynamic_threads` značí povoleno.

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

- Nastavuje a vrací, zda je povolen vnořený paralelismus.
- Pokud není povoleno, vnořené paralelní bloky jsou serializovány.

OpenMP knihovní funkce – Mutexy

```
void omp_init_lock (omp_lock_t *lock)
void omp_destroy_lock (omp_lock_t *lock)
void omp_set_lock (omp_lock_t *lock)
void omp_unset_lock (omp_lock_t *lock)
int  omp_test_lock (omp_lock_t *lock)

void omp_init_nest_lock (omp_nest_lock_t *lock)
void omp_destroy_nest_lock (omp_nest_lock_t *lock)
void omp_set_nest_lock (omp_nest_lock_t *lock)
void omp_unset_nest_lock (omp_nest_lock_t *lock)
int  omp_test_nest_lock (omp_nest_lock_t *lock)
```

- Inicializuje, ničí, blokujícně čeká, odemyká a testuje –
- – normální a rekurzivní mutex.

Proměnné prostředí

`OMP_NUM_THREADS`

- Specifikuje defaultní počet vláken, který se vytvoří při použití direktivy `parallel`.

`OMP_DYNAMIC`

- Hodnota `TRUE`, umožňuje za běhu měnit dynamicky počet vláken.

`OMP_NESTED`

- Povoluje hodnotou `TRUE` vnořený paralelismus.
- Hodnotou `FALSE` specifikuje, že vnořené paralelní konstrukce budou serializovány.

`OMP_SCHEDULE`

- Udává defaultní nastavení mapování iterací cyklu na vlákna.
- Příklady hodnot: "`static,4`", `dynamic`, `guided`.

Intel's Thread Building Blocks (TBB)

Thread Building Blocks

Co je Intel TBB

- TBB je C++ knihovna pro vytváření vícevláknových aplikací.
- Založená na principu zvaném Generic Programming.
- Vyvinuto synergickým spojením Pragma direktiv (OpenMP), standardní knihovny šablon (STL, STAPL) a programovacích jazyků podporující práci s vlákny (Threaded-C, Cilk).

Generic Programming

- Vytváření aplikací specializací existujících předpřipravených obecných konstrukcí, objektů a algoritmů.
- Lze nalézt v objektově orientovaných jazycích (C++, JAVA).
- V C++ jsou obecnou konstrukcí šablony (templates).
  - `Queue<Int>`
  - `Queue<Queue<Char»`

Použití TBB

Vlastnosti Intel TBB

- Knihovna, implementovaná s využitím standardního C++.
- Nepožaduje podporu speciálního jazyka či překladače.
- Podporuje vnořený paralelismus, potažmo je možné stavět složitější paralelní systémy z menších paralelních komponent.
- Cílem použití je nechat programátora specifikovat úlohy k paralelnímu provedení, nikoliv ho nutit popisovat, co a jak dělají jednotlivá vlákna.

Home Page

- `http://www.threadingbuildingblocks.org/`

Možnosti TBB

TBB poskytuje šablony pro

- Paralelizaci iterací jednotlivých cyklů – datový paralelismus.
- Definici vlastních paralelně přistupovaných datových struktur.
- Využití nízkoúrovňových HW primitiv.
- Zamykání přístupů do kritické sekce v různých podobách.
- Snadnou definici paralelních souběžných úloh.
- Škálovatelnou alokaci paměti.

IB109

- Pouze demonstrace použití TBB.
- Kompletní použití TBB je nad rámec tohoto kurzu.

Princip datové paralelizace v TBB

Paralelní for-cyklus

- Je dána množina nezávislých indexů, tzv. rozsah (range).
- Pro každý index z množiny je provedeno tělo cyklu.

Paralelní for-cyklus v TBB

- Šablona, která má dva parametry – Rozsah a tělo cyklu.
- Šablona zajistí vykonání těla cyklu pro všechny indexy ve specifikovaném rozsahu.
- Rozsah je dělen na pod-rozsahy. Paralelismu dosaženo souběžným vykonáváním těla cyklu nad jednotlivými pod-rozsahy.

Příklad – paralelní for

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

const int n=1000;
float input[n];
float output[n];

struct Average {
   void operator()( const blocked_range<int>& range ) const {
     for( int i=range.begin(); i!=range.end(); ++i )
       output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.f);
     }
};

Average avg;

parallel_for( blocked_range<int>( 1, n ), avg );
```

Koncept dělení

Koncept dělení

- Instance některých tříd je nutné za běhu (rekurzivně) dělit.
- Zavádí se nový typ konstruktoru, dělicí konstruktor:

```
X::X(X& x, split)
```

- Dělicí konstruktor rozdělí instanci třídy `X` na dvě části, které dohromady dávají původní objekt. Jedna část je přiřazena do `x`, druhá část je přiřazena do nově vzniklé instance.
- Schopnost dělit-se musí mít zejména rozsahy, ale také třídy, jejichž instance běží paralelně a přitom nějakým způsobem interagují, např. třídy realizující paralelní redukci.

`split`

- Speciální třída definovaná za účelem odlišení dělicího konstruktoru od kopírovacího konstruktoru.

Koncept rozsahu

Požadavky na třídu realizující rozsah

- Kopírovací konstruktor

  ```
  R::R (const R&)
  ```

- Dělicí konstruktor

  ```
  R::R (const R&, split)
  ```

- Destruktor

  ```
  R::~R ()
  ```

- Test na prázdnost rozsahu

  ```
  bool R::empty() const
  ```

- Test na schopnost dalšího rozdělení

  ```
  bool R::is_divisible() const
  ```

Předdefinované šablony rozsahů

- Jednodimenzionální: `blocked_range`
- Dvoudimenzionální: `blocked_range2d`

TBB: blocked_range

`blocked_range`

- `template<typename Value> class blocked_range;`
- Reprezentuje nadále dělitelný otevřený interval `[i,j)`.

Požadavky na třídu `Value` specializující `blocked_range`

- Kopírovací konstruktor

  ```
  Value::Value (const Value&)
  ```

- Destruktor

  ```
  Value::~Value ()
  ```

- Operátor porovnání

  ```
  bool Value::operator<(const Value& i, const Value& j)
  ```

- Počet objektů v daném rozsahu (operátor $-$)

  ```
  size_t Value::operator-(const Value& i, const Value& j)
  ```

- k-tý následný objekt po i (operátor $+$)

  ```
  Value Value::operator+(const Value& i)
  ```

TBB: blocked_range

Použití `blocked_range<Value>`

- Nejdůležitější metodou je konstruktor.
- Konstruktor specifikuje interval rozsahu a velikost největšího dále nedělitelného sub-intervalu:
- `blocked_range(Value begin, Value end [, size_t grainsize] )`

Typická specializace

- `blocked_range<int>`
- Příklad: `blocked_range<int>(5, 17, 2)`
- Příklad: `blocked_range<int>(0, 11)`

TBB: parallel_for

`parallel_for<Range,Body>`

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

Požadavky na třídu realizující tělo cyklu

- Kopírovací konstruktor

  ```
  Body::Body (const Body&)
  ```

- Destruktor

  ```
  Body::~Body ()
  ```

- Aplikátor těla cyklu na daný rozsah – operátor ()

  ```
  void Body::operator()(Range& range) const
  ```
  ```

TBB: parallel_reduce

`parallel_reduce<Range,Body>`

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

Požadavky na třídu realizující tělo redukce

- Dělicí konstruktor

  `Body::Body (const Body&, split)`

- Destruktor

  `Body::~Body ()`

- Funkce realizující redukci nad daným rozsahem – operátor ()

  `void Body::operator()(Range& range)`

- Funkce realizující redukci hodnot z různých rozsahů

  `void Body::join(Body& to_be_joined)`

Možnosti dělení

Třída `Partitioner`

- Paralelní konstrukce mají třetí volitelný parametr, který specifikuje strategii dělení rozsahu.
- `parallel_for<Range,Body,Partitioner>`

Předdefinované strategie

- `simple_partitioner`
  - Rekurzivně dělí rozsah až na dále nedělitelné intervaly.
  - Při použití `blocked_range` je volba `grainsize` klíčová pro vyvážení potenciálu a režie paralelizace.
- `auto_partitioner`
  - Automatické dělení, které zohledňuje zatížení vláken.
  - Při použití `blocked_range` volí rozsahy větší, než je `grainsize` a tyto dělí pouze do té doby, než je dosaženo rozumného vyvážení zátěže. Volba minimální velikosti `grainsize` nezpůsobí nadbytečnou režii spojenou s paralelizací.

Paralelně přistupované kontejnery – `vector` a `queue`

`concurrent_queue`

- `template<typename T> concurret_queue`
- Fronta, ke které může souběžně přistupovat více vláken.
- Velikost fronty je dána počtem operací vložení bez počtu operací výběru. Záporná hodnota značí čekající operace výběru.
- Definuje sekvenční. iterátory, nedoporučuje se je používat.

`concurrent_vector`

- `tempate<typename T> concurrent_vector`
- Zvětšovatelné pole prvků, ke kterému je možné souběžně přistupovat z více vláken a provádět souběžně zvětšování pole a přístup k již uloženým prvkům.
- Nad vektorem lze definovat rozsah a provádět skrze něj paralelně operace s prvky uloženými v poli.

Paralelně přistupované kontejnery – `hash_map`

`concurrent_hash_map`

- `template<typename Key, typename T, typename HashCompare>`
  `class concurrent_hash_map;`
- Mapa, ve které je možné paralelně hledat, mazat a vkládat.

Požadavky na třídu `HashCompare`

- Kopírovací konstruktor

  ```
  HashCompare::HashCompare (const HashCompare&)
  ```

- Destruktor

  ```
  HashCompare::~HashCompare ()
  ```

- Test na ekvivalenci objektů

  ```
  bool HashCompare::equal(const Key& i, const Key& j)const
  ```

- Výpočet hodnoty hešovací funkce

  ```
  size_t HashCompare::hash(const Key& k)
  ```

Paralelně přistupované kontejnery – `hash_map`

Objekty pro přístup k datům v `concurrent_hash_map`

- Přístup k párům Klíč-Hodnota je skrze přistupovací třídy.
- `accessor` – pro přístup v režimu read/write
- `const_accessor` – pro přístup pouze v režimu read
- Použití přistupovacích objektů umožňuje korektní paralelní přístup ke sdíleným datům.

Příklad použití přistupovacího objektu

```
typedef concurrent_hash_map<Int,Int> MyTable;
MyTable table;

MyTable::accessor a;
table.insert( a, 4 );
a->second += 1;
a.release();
```

Paralelně přistupované kontejnery – `hash_map`

Metody pro práci s `concurrent_hash_map`

- `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)`

Další způsoby použití

- Iterátory pro procházení mapy.
- Lze definovat rozsahy a s nimi pracovat paralelně.

Sekce

C++11

C++11 a vláknování

Pozorování

- C++11 má definované příkazy pro podporu vláken.
- Není třeba používat externí knihovny jako je POSIX Thread.

Jak je to možné

- C++11 definuje virtuální výpočetní stroj.
- Veškerá sémantika příkazů se odkazuje na tento virtuální výpočetní stroj.
- Virtuální výpočetní stroje je paralelní, příkazy související s podporou vláken mohou bý součástí jazyka.
- Přenos sémantiky z virtuálního výpočetního stroje na reálný HW je na zodpovědnosti překladače.

Příklad – Vlákna a mutexy v C++11

```
#include <thread>
#include <mutex>
std::mutex mylock;

void func(int& a)
{
   mylock.lock();
   a++;
   mylock.unlock();
}

int main()
{
   int a = 42;
   std::thread t1(func, std::ref(a));
   std::thread t2(func, std::ref(a));
   t1.join();
   t2.join();
   std::cout « a « std::endl;
   return 0;
}
```

Zamykání v C++11

Potencionální riziko uváznutí

- Jazyk s plnou podporou mechanismu výjimek.
- Vyvolání výjimky v okamžiku, kdy je vlákno v kritické sekci (uvnitř mutexu) pravděpodobně způsobí, že nebude vláknem volána metoda odemykající zámek svázaný s kritickou sekcí.

Řešení

- Využití principu RAII a OOP.
- Zamčení mutexu realizováno vytvořením lokální instance vhodné předdefinované zamykací třídy.
- Odemykání umístěno do destruktoru této třídy.
- Destruktor je proveden v okamžiku opuštění rozsahu platnosti daného objektu.

RAII zamykání v C++11

Třída `lock_guard`

- Obalení standardního zámku v RAII stylu.
- Mutex na pozadí nelze „předat“ jinému vláknu, nevhodné pro podmínkové proměnné.
- Příklad použití:

```
std::mutex m;
void func(int& a)
{
  std::lock_guard<std::mutex> l(m);
  a++;
}
```

Třída `unique_lock`

- Obecnější předatelné RAII obalení mutexu.
- Doporučené pro použití s podmínkovými proměnnými.

Podporované aspekty

Podpora vláknování v C++11

- Vlákna.
- Mutexy a RAII zámky.
- Podmínkové proměnné.
- Sdílené futures (místa uložení dosud nespočítané hodnoty).

Rozcestník

- `http://en.cppreference.com/w/cpp/thread`

Jiné rychlé přehledy

- `http://www.codeproject.com/Articles/598695/Cplusplus-threads-locks-and-condition-variables`
- `http://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g`

Atomicita zápisů

Neatomicky

- `int x,y;`

```
Thread 1                Thread 2
x = 17;                 cout « y « " ";
y = 37;                 cout « x « endl;
```

- Nemá definované chování.

Správně atomicky

- `atomic<int> x, y;`

```
Thread 1                Thread 2
x.store(17);            cout « y.load() « " ";
y.store(37);            cout « x.load() « endl;
```

- Chování je definované, možné výstupy: 0 0, 0 17, 37 17.

Práce s paměťovým modelem v C++11

Paměťový model

- Implicitní chování zachovává sekvenční konzistenci (automaticky vkládá odpovídající paměťové bariéry).
- Riziko neefektivního kódu.

Příklad 1

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

  Thread 1
  x.store(17,memory_order_relaxed);
  y.store(37,memory_order_relaxed);

  Thread 2
  cout « y.load(memory_order_relaxed) « " ";
  cout « x.load(memory_order_relaxed) « endl;
  ```

- Sémantika povoluje v tomto případě i výstup: 37 0.
  ```

Práce s paměťovým modelem v C++11 – pokračování

Paměťový model

- Implicitní chování zachovává sekvenční konzistenci (automaticky vkládá odpovídající paměťové bariéry).
- Riziko neefektivního kódu.

Příklad 2

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

  Thread 1
  x.store(17,memory_order_release);
  y.store(37,memory_order_release);

  Thread 2
  cout << y.load(memory_order_acquire) << " ";
  cout << x.load(memory_order_acquire) << endl;
  ```

- Acquire nepřeuspořádá operace `load`, Release – `store`.
  ```

Jiné přístupy

Paralelní for cyklus

Paralelní for cyklus

- Nejčastější a nejednoduší metoda paralelizace.
- Datová paralelizace.

Jak a kde lze řešit paralelní for cyklus

- `http://parallel-for.sourceforge.net/`

IB109 Návrh a implementace paralelních systémů

Principy návrhu paralelních algoritmů

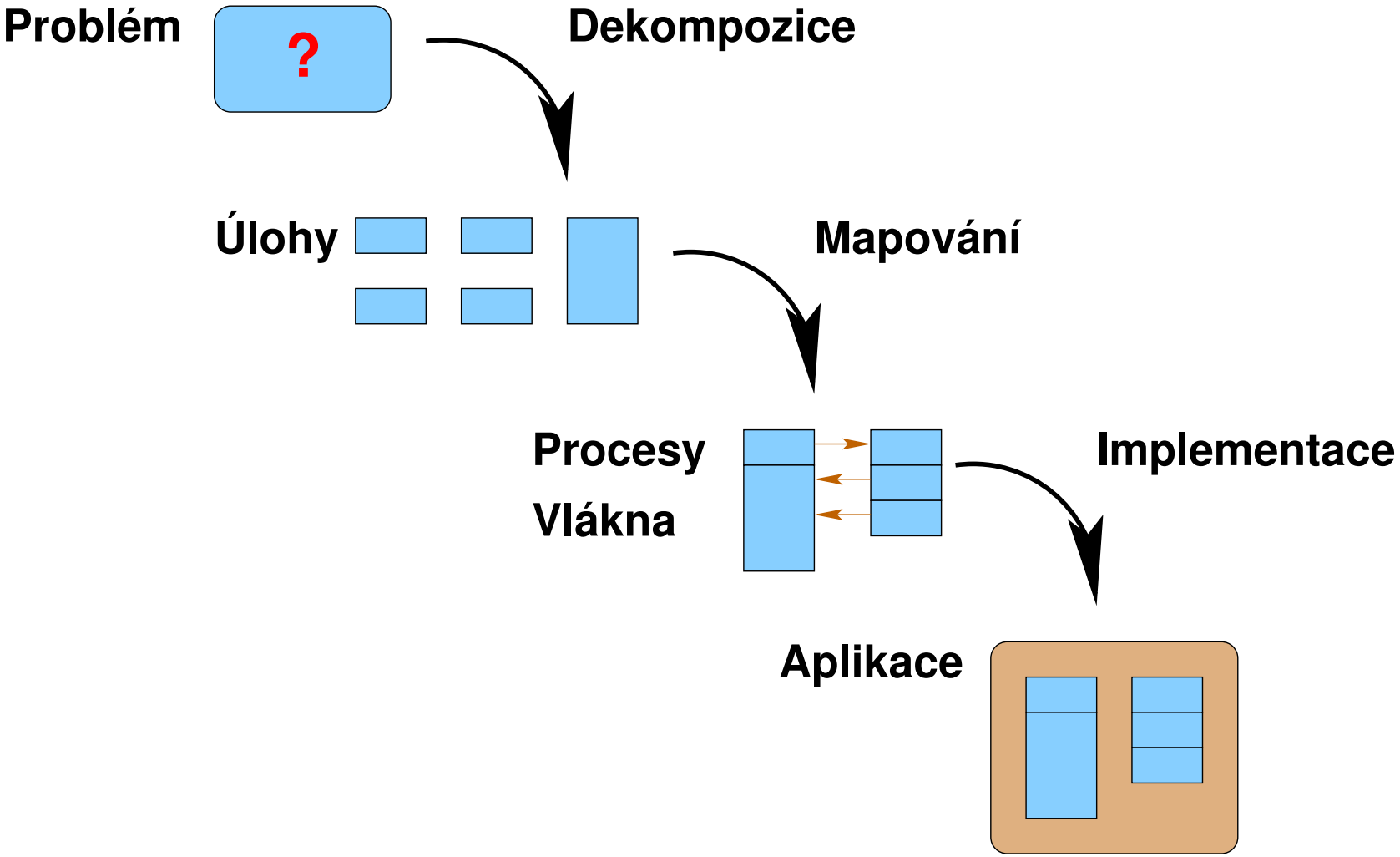
Jiří Barnat

Více-práce programátora paralelních aplikací

- Identifikovat souběžně proveditelné činnosti a jejich závislosti.
- Mapovat souběžně proveditelné části práce do procesů.
- Zajistit distribuci vstupních, vnitřních a výstupních dat.
- Spravovat souběžný přístup k datům a sdíleným prostředkům.
- Synchronizovat jednotlivé procesy v různých stádiích výpočtu tak, jak vyžaduje paralelní algoritmus.
- Mít znalost přídavných programátorských prostředků související s vývojem paralelních algoritmů.

Základy návrhu paralelních algoritmů

Návrh a realizace paralelního systému



Dekompozice a Úlohy

Dekompozice

- Proces rozdělení celé výpočetní úlohy na podúlohy.
- Některé podúlohy mohou být prováděny paralelně.

(Pod)úlohy

- Jednotky výpočtu získané dekompozicí.
- Po vyčlenění se považují za dále nedělitelné.
- Mají uniformní/neuniformní velikost.
- Jsou definované v době kompilace / za běhu programu.

Příklad

- Násobení matice A $(n \times n)$ vektorem B
- $C[i] = \sum_{j=1}^{n} A[i,j].B[j]$

Závislosti úloh

Graf závislostí

- Zachycuje závislosti prováděných úloh.
- Definuje relativní pořadí provádění úloh (částečné uspořádání).

Vlastnosti a využití grafu

- Orientovaný acyklický graf.
- Graf může být nespojitý, či dokonce prázdný.
- Úloha je připravena ke spuštění, pokud úlohy, na kterých závisí, dokončily svůj výpočet (topologické uspořádání).

Příklady závislostí

- Pořadí oblékání svršků.
- Paralelní vyhodnocování výrazů

  `v AND x AND (y OR z)`

Granularita a Stupeň souběžnosti

Granularita

- Počet úloh, na které se problém dekomponuje.
- Mnoho malých úloh – jemnozrnná granularita (fine-grained).
- Málo větších úloh – hrubozrnná granularita (coarse-grained).
- Každý problém má vnitřní hranici granularity.

Stupeň souběžnosti

- Maximální počet úloh, které mohou být prováděny souběžně.
- Limitem je vnitřní hranice granularity.

Průměrný stupeň souběžnosti

Průměrný stupeň souběžnosti

- Závislý na grafu závislostí a granularitě.
- Mějme množství práce asociované k uzlům grafu.
- **Kritická cesta** – cesta, na které je součet prací maximální.
- **Průměrný stupeň souběžnosti** je podíl celkového množství práce vůči množství práce na kritické cestě.
- Udává maximální zrychlení, pokud je cílová platforma schopna vykonávat souběžně maximální stupeň souběžnosti úloh.

Pozorování

- Zjemňování dekompozice může zvýšit stupeň souběžnosti.
- Čím méně práce je na kritické cestě, tím větší je potenciál pro paralelizaci.

Interakce úloh – Další omezení paralelizace

Interakce úloh

- Nezávislé úlohy mohou vzájemně komunikovat.
- Obousměrná komunikace může snižovat stupeň souběžnosti (úlohy musí co-existovat ve stejný okamžik).
- Komunikace úloh – **neorientovaný graf interakcí**.
- Graf interakcí pokrývá graf závislostí (ověření splnění předpokladů pro spuštění úlohy je forma interakce).

Příklad jednosměrné komunikace

- Násobení matice vektorem ($y = Ab$)
- Dekompozice na nezávislé úlohy dle řádků matice A.
- Prvky vektoru b jsou čteny ze všech úloh, je nutné je vhodně distribuovat k jednotlivým úlohám.

Techniky dekompozice

Techniky dekompozice

Dekompozice

- Fundamentální technika v návrhu paralelních algoritmů.

Obecné dekompozice

- Rekurzivní
- Datová

Specializované dekompozice

- Průzkumová
- Spekulativní
- Hybridní

Rekurzivní dekompozice

Vhodné pro problémy typu rozděl a panuj.

Princip

- Problém se dekomponuje na podúlohy tak, aby jednotlivé úlohy mohly být dekomponovány stejným způsobem jako rodičovská úloha.
- Někdy je třeba restrukturalizovat úlohu.

Příklad

- Quicksort
  - Provede se volba pivota.
  - Rozdělení pole na prvky menší než a větší rovno než.
  - Rekurzivně se opakuje dokud je množina prvků neprázdná.
- Hledání minima v lineárním poli.
  - Princip půlení prohledávaného pole.
  - Typický příklad restrukturalizace výpočtu.

Datová dekompozice

Základní princip

- Data se rozdělí na části (data partitioning).
- Úlohy se provádí souběžně nad jednotlivými částmi dat.

Datová dekompozice podle místa

- Vstupní data
- Výstupní data
- Vnitřní data
- Kombinace

Mapování dat na úlohy

- Funkce identifikující vlákno odpovědné za zpracování dat.

Úlohy typu "Embarrassingly parallel"

- Triviální datová dekompozice na dostatečný počet zcela nezávislých, vzájemně nekomunikujících úloh.

Průzkumová dekompozice

Princip

- Specializovaná technika paralelizace.
- Vhodná pro prohledávací úlohy.
- Prohledávaný prostor se rozdělí podle směru hledání.

Vlastnosti

- Při znalosti prohledávaného stavového prostoru lze dosáhnout optimálního vyvážení a zatížení procesorů.
- Na rozdíl od datové dekompozice, úloha končí jakmile je nalezeno požadované.
- **Množství provedené práce se liší od sekvenční verze.**
- V případě, že graf není strom, je třeba řešit problém opakujících se konfigurací (riziko nekonečného výpočtu).

Příklad

- Řešení hlavolamu “patnáct”

Spekulativní dekompozice

Princip

- Specializovaná technika paralelizace.
- Vhodná pro úlohy se sekvencí datově závislých podúloh.
- Úloha, která čeká na výstup předchozí úlohy, se spustí nad všemi možnými vstupy (výstupy předchozí úlohy).

Vlastnosti

- Provádí se zbytečná práce.
- Nemusí být ve výsledku rychlější jak serializovaná verze.
- Vhodné pro úlohy, kde jistá hodnota mezivýsledku má velkou pravděpodobnost.
- Vzniká potenciální problém při přístupu ke zdrojům (některé zdroje nemusí být sdílené v případě sekvenčního vykonávání úloh).

Příklady

- Spekulativní provádění kódu (větvení).

Hybridní dekompozice

Kombinace různých způsobů dekompozice

Příklad

- Hledání minima v poli.
- Sekvenční verze najde minimum v $O(n)$.
- Při použití datové a rekurzivní dekompozice lze trvání této úlohy zkrátit na $O(n/p + log(p))$.
- Vstupní pole se datově dekomponuje na p stejných částí.
- Najdou se minima v jednotlivých částech v čase $O(n/p)$.
- Výsledky z jednotlivých třídění se zkombinují v čase $O(log(p))$.
- Teoreticky lze při dostatečném počtu procesorů nalézt minimum v čase $O(log(n))$.
- Tento styl paralelismu je obecně označován jako **MAP-REDUCE**.

Techniky mapování a vyrovnávání zátěže

Mapování úloh na vlákna/procesy.

Mapování

- Přiřazování úloh jednotlivým vláknům/procesům.
- Optimální mapování bere v potaz grafy závislostí a interakce.
- Ovlivňuje výkon aplikace.
- Naivní mapování (úloha=proces/vlákno)

Cíle mapování

- Minimalizovat celkový čas řešení celé úlohy.
  - Redukovat prodlevy způsobené čekáním (idling)
  - Redukovat zátěž způsobenou interakcí
  - Redukovat režii spouštění, ukončování a přepínání
  - Vyrovnat zátěž na jednotlivé procesory
- Maximalizovat souběžnost.
- Minimalizovat zatížení systému (zatížení datových cest).
- Využít dostupnost zdrojů použitých předchozí úlohou.

Charakteristiky úloh, které ovlivňují mapování

Způsob zadání úlohy

- **Statické zadání úloh** – dekompozice problému na úlohy je dána v době kompilace, případně je přímo odvozena od vstupních dat.
- **Dynamické zadání úloh** – nové úlohy jsou vytvářeny za běhu aplikace dle průběhu výpočtu, případně jako důsledek provádění původně zadaných úloh.

Velikost úlohy

- Relativní množství času potřebné k dokončení úlohy.
- Uniformní vs. neuniformní.
- Dopředná znalost/neznalost.

Velikost dat asociovaných k úloze

- Snaha o zachování lokality dat.
- Různá data mají různou roli a velikost (vstupní/výstupní data u hlavolamu patnáct).

Charakteristiky interakcí, které ovlivňují mapování

Statické vs Dynamické

- Statické: Probíhají v předdefinovaném časovém intervalu, mezi předem známou množinou úloh.
- Dynamické: Pokud předem neznáme počet interakcí, časový rámec interakcí, nebo participující úlohy.

Další charakteristiky

- Jednosměrná versus obousměrná interakce.
- Mód přístupu k datům: Read-Only versus Read-Write.
- Pravidelné versus nahodilé interakce.

Režie související s mapováním do různých vláken/procesů

- Uzpůsobení aplikace pro neočekávanou interakci.
- Připravenost dat k odeslání / adresáta k přijetí.
- Řízení přístupu ke sdíleným zdrojům.
- Optimalizace aplikace pro redukci prodlev.

Schémata pro Statické Mapování

Mapování založené na rozdělení dat

- Bloková distribuce
- Cyklická a blokově-cyklická distribuce
- Náhodná distribuce bloků
- Dělení grafu

Mapování založené na rozdělení úloh

- Dělení dle grafu závislostí úloh
- Hierarchické dělení

Mapování založené na rozdělení dat

Bloková distribuce datových polí

- Procesy svázány s daty rozdělenými na souvislé bloky.
- Bloky mohou být vícerozměrné (redukce interakcí).
- Příklad
  - Násobení matic $A \times B = C$
  - Dělení matice C na 1- a 2-rozměrné bloky.

Cyklická a blokově-cyklická distribuce datových polí

- Nerovnoměrné množství práce spojené s jednotlivými prvky
- $\Rightarrow$ blokové dělení způsobuje nerovnoměrné zatížení.
- Blokově-cyklická distribuce: dělení na menší díly a cyklické přiřazení procesům (round robin).
- Zmenšování bloků vede k cyklické distribuci (blok je atomický prvek datového pole).

Mapování založené na rozdělení dat – pokračování

Náhodná distribuce bloků

- Zátěž související s prvky pole vytváří pravidelné vzory.
- $\Rightarrow$ špatná distribuce v cyklickém rozdělení.
- Náhodné přiřazení bloků procesům.

Grafové dělení

- Pro případy, kdy je nevhodné organizovat data do polí (například drátové modely 3D objektů).
- Data organizována jako graf.
- Optimální dělení.
  - Stejný počet vrcholů v jednotlivých částech.
  - Co možná nejmenší počet hran mezi jednotlivými částmi.
  - NP-úplný problém.

Mapování založené na rozdělení úloh

Princip

- Graf závislostí úloh.
- Grafové dělení (NP-úplné).

Speciální případy pro konkrétní tvar grafů

- Binární strom (rekurzivní dekompozice).

Hierarchické mapování

- Úlohové dělení nebere v potaz neuniformitu úloh.
- Shlukování úloh do nad-úloh.
- Definuje hierarchie (vrstvy).
- Jiné mapovací a dekompoziční techniky na jednotlivých vrstvách.

Schémata pro Dynamické Mapování

Motivace

- Statické mapování nedostatečné, neboť charakteristiky úloh nejsou známy v době překladu.

Centralizovaná schémata dynamického mapování

- Úlohy jsou shromažďovány v jednom místě.
- Dedikovaná úloha pro přiřazování úloh procesům.
- Samo-plánování
  - Jakmile proces dokončí úlohu, vezme si další.
- Blokové plánování
  - Přístup ke shromaždišti úloh může být úzkým místem,
  - $\Rightarrow$ přidělování úloh po blocích.

Příklad

- Třídění prvků v $n \times n$ matici A
- `for (i=1; i<n; i++) newtask(sort(A[i],n));`

Schémata pro Dynamické Mapování – pokračování

Distribuovaná schémata

- Množina úloh je distribuována mezi procesy.
- Za běhu dochází k vzájemnému vyměňování úloh.
- Netrpí nedostatky spojenými s centralizovaným řešením.

Možnosti

- Jak se určí, kdo komu pošle úlohu.
- Kdo a na základě čeho určí, že je potřeba přesunout úlohu.
- Kolik úloh má být přesunuto.
- Kdy a jak je úloha přesunuta.

Problém

- Efektivita přenosu úlohy na jiný proces.

Afinitní plánování

Vlákna a procesory

- Jednotlivá vlákna jsou vykonávány fyzickými procesory.
- Plánování zajišťuje plánovač OS.

Afinitní plánování (angl. affinity scheduling)

- Modifikace algoritmu plánování.
- Afinitní plánování zajišťuje, že výpočetní dávky přidělené jednomu procesu/vláknu budou pokud možno přiděleny na fyzicky tentýž procesor.

Výhody a rizika

- Potencionálně lepší využití cache.
- Striktní lpění na tomtéž procesoru může narušovat vyváženost využití procesorů, tedy redukovat výkon aplikace.

Dilema procesu dekompozice-mapování

Otázka

- Je lepší nejprve dekomponovat na mnoho malých úloh a pak úlohy shlukovat při mapování, nebo naopak omezit dekompozici, aby mapování bylo přímočaré?

Aspekty napomáhající rozhodnutí dilematu

- Je cena dekompozice shodná pro oba scénáře?
- Vytváří jemnější dekompozice skutečně nezávislé úlohy?
- Je jemnější dekompozicí zachována datová lokalita?
- Je/není znám počet jader na cílové platformě?
- Jaká je cena režie přepínání, zejména v situaci, kdy počet vláken výrazně převyšuje počet výpočetních jader?

Metody pro redukci režie interakce

Metody pro redukci režie interakce

Režie související s interakcí

- Režie související s interakcí souběžných úloh je klíčovým faktorem ovlivňujícím výkon paralelní aplikace.
- Z pohledu režie interakce jsou ideální "Embarrassingly parallel"úlohy, kde k interakci nedochází.

Faktory ovlivňující režii

- Objem přenášených dat
- Frekvence interakce
- Cena komunikace

Zvyšování datové lokality

Cíl – snížit objem přenášených dat

- Přesun sdílených datových struktur do lokálních kopií.
- Minimalizace objemu sdílených dat.
- Lokalizace výpočtu (lokální ukládání mezivýsledků).
- Režie protokolů pro udržení koherence lokálních kopií.

Cíl – snížit frekvenci interakce

- Prostorová lokalizace přenášených dat
- Přenášení dat a jejich okolí (princip cache)
- Více zpráv v jedné (bufferování)

Minimalizace současných přístupů

Problém – Contention

- Přístup k omezenému zdroji ve stejný okamžik (contention) je řešen serializací požadavků.
- Serializace požadavků způsobuje prodlevy.

Možné řešení

- Je potřeba N souběžných přístupů k datům.
- Přistupovaná data je možné rozdělit do N bloků.
- A data číst v N po sobě jdoucích iteracích.
- V každé iteraci je každý blok čten jiným vláknem.
- Číslo čteného bloku v r-té iteraci j-tým vláknem:
  $(r + j) modulo\ N$

Překrývání výpočtu s interakcí

Problém

- Čekání na příjem či odeslání dat způsobuje nechtěné prodlevy.

Včasné vykonání akce – podmínky proveditelnosti

- Data musí být včas připravena.
- Přijímací i odesílací strany mohou asynchronně komunikovat.
- Existuje další úloha, která může být řešena po dobu komunikace.

Jiná řešení

- Simulace mechanismu přerušení (ala operační systém).
- Žádné, kvůli režii způsobené násilným řešením.

Replikace dat či výpočtů

Problém

- Opakované drahé přístupy ke sdíleným datům.

Řešení pro read-only data

- Při prvotní interakci tvorba kopii dat (datová lokalita).
- Dále pracovat s lokální kopií.
- Zvyšuje paměťové nároky výpočtu.

Řešení pro read-write data

- Podobně jako v read-only případě.
- Násobné souběžné výpočty téhož mohou být rychlejší, než čtení a zápis sdílené hodnoty.

Optimalizované operace pro kolektivní komunikaci

Problém

- Stejná interakce mezi všemi procesy vykonávaná základními komunikačními primitivy je drahá.

Řešení – kolektivní komunikační operace

- Pro přístup k datům jiných vláken/procesů.
- Důležité pro komunikačně intenzivní výpočty.
- Forma efektivní synchronizace.

Optimalizované implementace

- MPI

Překrývání Interakcí

Problém

- Nedostatečná propustnost komunikační sítě, či absence kolektivních komunikačních operací.

Řešení

- Zvýšit využití komunikační sítě současnou komunikací mezi různými páry procesů.

Příklad

- 4 procesy $P_1, \ldots, P_4$
- P_1 chce všem poslat zprávu m_1
- $P_1 \rightarrow P_2$, $P_2 \rightarrow P_3$, $P_3 \rightarrow P_4$
- $P_1 \rightarrow P_2$, $P_2 \rightarrow P_3$

  $P_1 \rightarrow P_4$

Komunikace v nesdíleném adresovém prostoru

- Synchronizace posíláním zpráv.
- Předávání dat posíláním zpráv.

Komunikace ve sdíleném adresovém prostoru

- Synchronizace korektním přístupem ke sdíleným datům.
- Předávání dat pomocí sdílených datových struktur. (FIFO)

Obecné charakteristiky

- Latence – doba potřebná pro doručení prvního bitu.
- Přenosová rychlost – objem dat přenesených za jednotku času.

Latence

- Celková cena pro zahájení komunikace — t_s
  - čas pro přípravu zprávy/dat
  - identifikace adresáta / routování
  - doba trvání vylití informace z cache do paměti případně na síťové rozhraní
- Cena "hopů"(přeposílání uvnitř komunikační sítě) — t_h
  - čas strávený na jednotlivých routerech v síti
  - doba, po kterou putuje hlavička zprávy z přijímacího na odesílací port

Přenosová rychlost

- Ovlivněno šířkou pásma r
- Cena za přenos jednoho slova (word = 2 bajty) — $t_w = 1/r$

Cena komunikace

- m – délka zprávy ve slovech
- l – počet linek, přes které zpráva putuje
- $t_s + l * (t_h + mt_w)$

Obecné metody redukce ceny

- Spojování malých zpráv (amortizuje se hodnota t_s)
- Komprese (snižování hodnoty m)
- Minimalizace vzdálenosti (snižování hodnoty l)
- Paketování (eliminace režie způsobené jednotlivými hopy)

$$t_s + l * (t_h + mt_w) \quad \longrightarrow \quad t_s + lt_h + mt_w$$

IB109 Návrh a implementace paralelních systémů

Kolektivní komunikační primitiva

Jiří Barnat

Kvantitativní parametry komunikace

Komunikace (interakce)

- Předávání informací mezi jednotlivými procesy

Parametry komunikace

- Latence — zpoždění související se započetím vlastní komunikace
- Šířka pásma (bandwidth) — maximální množství dat přenesených za jednotku času
- Objem — množství předávaných dat

Cena komunikace

- t_s – latence, t_w – šířka pásma, m – objem
- $t_s + m * t_w$

Kvantitativní parametry komunikace

Příklady

- Za jak dlouho napustím hrníček vodou pomocí 2km dlouhé zahradní hadice?

- Při konstantním čtení z paměti trvá získání kódu instrukce a příslušných operandů z paměti v průměru 5ns. Jaká je nejvyšší reálná rychlost vykonávání kódu uloženého v paměti 4GHz procesorem?

  $[\ 1/(5 * 10^{-9}) = 0.2 * 10^{9} = 200\ \text{MHz}\ ]$

Efektivita interakce

Pozorování

- Interakce jednotlivých úloh/procesů je nevyhnutelná
- Interakce způsobuje prodlevy ve výpočtu
- Režie související s interakcí by neměla být důvodem pro neefektivitu paralelního zpracování

Závěr

- Komunikační primitiva musí být co nejefektivnější

V této přednášce

- Popis různých typů komunikačních operací
- Odvození ceny pro jednotlivé topologie

Topologie komunikační sítě

Topologie

- Fyzická/logická struktura komunikačních kanálů mezi jednotlivými participanty komunikace.

Vlastnosti komunikační sítě

- Průměr (délka maximální nejkratší cesty)
- Konektivita (minimální počet disjunktních cest)
- Stupeň (max. počet linek incidenčních s jedním vrcholem)
- Cena
- Výlučnost přístupu (použití jednou úlohou blokuje ostatní)
- Rozšiřitelnost
- Škálovatelnost

Sběrnice, Kliky

Sběrnice

- Sdílené médium
- Propustnost, klesá s počtem uzlů (je třeba cache)
- Důležitost: model sdíleného adresového prostoru

Kliky (úplné sítě)

- Privátní neblokující spojení každého s každým
- Cena: počet linek je kvadratický vůči N
- Cena: stupeň každého uzlu je $N - 1$
- Škálovatelné, za podmínky levného zvyšování stupně uzlu
- Důležitost: abstraktní představa sítě, logická struktura

Prsteny, Hvězdice

Prsten (kruh, řetěz, 1-rozměrná mřížka)

- Uspořádání na uzlech
- Privátní neblokující komunikace s 2 nejbližšími uzly
- Důležitost: logická struktura komunikace, Pipeline model

Hvězdice

- Spojení přes jeden centrální uzel
- Středový uzel může být úzkým místem
- Lze hierarchicky vrstvit (hvězdice hvězdic)
- Důležitost: Master-Slave model

Hyperkostka, Strom s aktivními vnitřními uzly

Hyperkostka

- Spojení n uzlů ve tvaru $log_2 n$-rozměrné krychle
- Stupeň uzlu je $log_2 n$
- Průměr sítě je $log_2 n$
- Počet linek $O(N \cdot log(N))$
- Postup konstrukce
  - binární označení uzlů s identifikátory 0 až $2^{log_2 n} - 1$
  - linka existuje mezi uzly pokud se označení liší v jednom bitu
  - minimální vzdálenost uzlů odpovídá počtu odlišných bitů v označení uzlů

Strom s aktivními vnitřními uzly

- Strom, kde participanty komunikace jsou listy i vnitřní uzly
- Kostra hyperkostky — Strom s maximální hloubkou $log_2 n$
- Důležitost: Optimální šíření informací

Další topologie

Jiné topologie

- Z hlediska logického návrhu paralelní aplikace nezajímavé.

Příklady

- Obecné stromy
- Přepojované sítě
- Vícevrstvé sítě (ω-síť)
- Mřížky
- Cyklické mřížky (torrus)

Jeden na všechny a všichni na jednoho

Jeden na všechny a všichni na jednoho

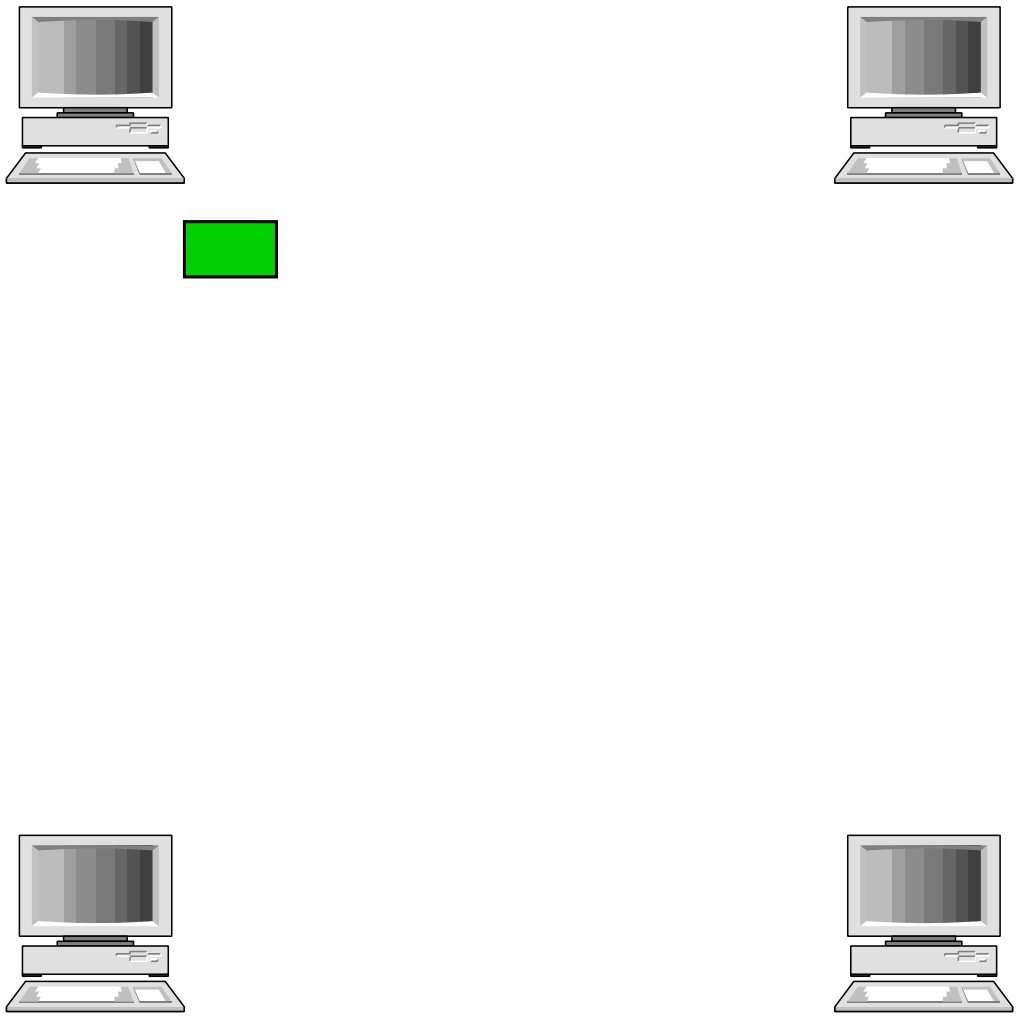
One-To-All Vysílání (OTA)

- Úloha posílá několika/všem ostatním identická data
- Ve výsledku je p kopií originální zprávy v lokálních adresových prostorech adresátů
- Změna struktury dat: $m \mapsto p * m$ (m-je velikost zprávy)

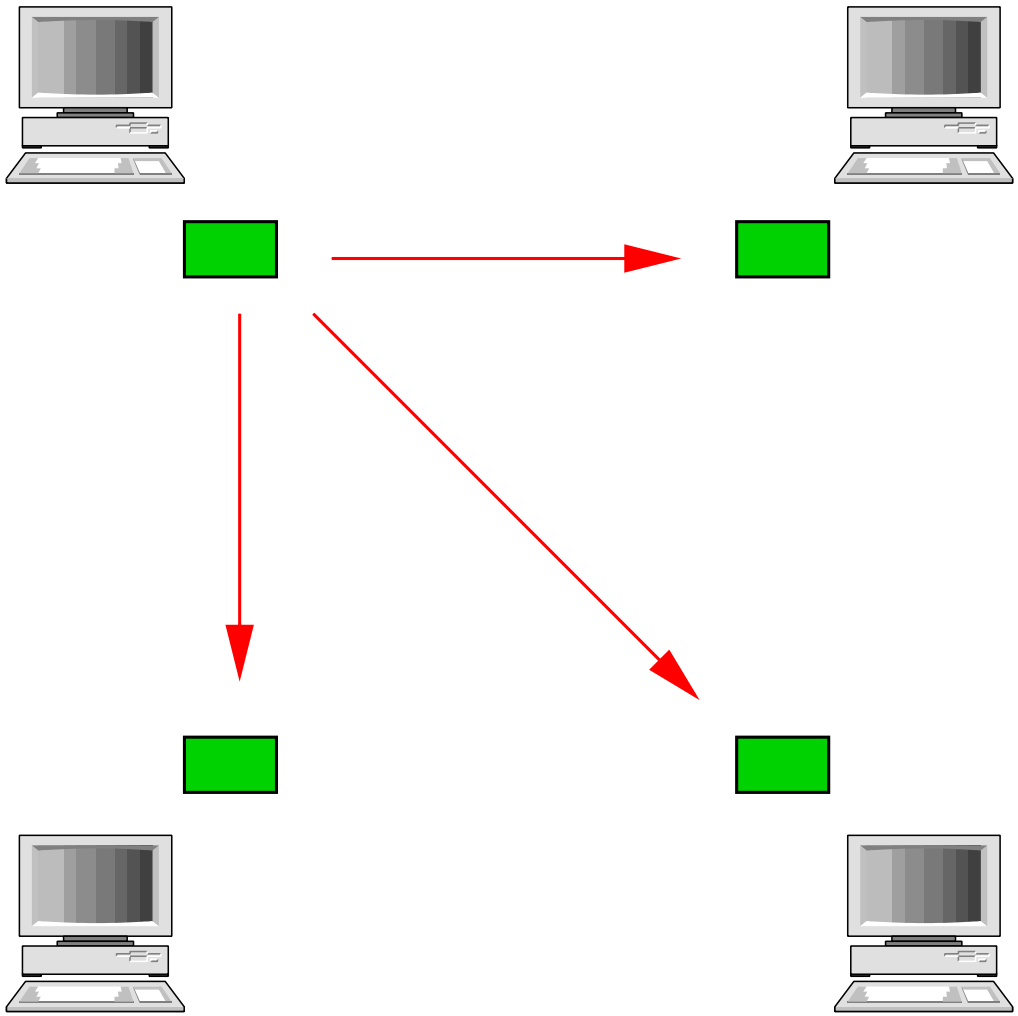
All-To-One Redukce (ATO)

- Duální operace k "One-To-All"
- Několik/všechny úlohy posílají data jedné úloze
- Data se kombinují pomocí zvoleného asociativního operátoru
- Ve výsledku je jedna kopie v adresovém prostoru cílové úlohy
- $m_1 \otimes m_2 \ldots \otimes m_p \rightsquigarrow m$
- Změna struktury dat: $m * p \mapsto m$

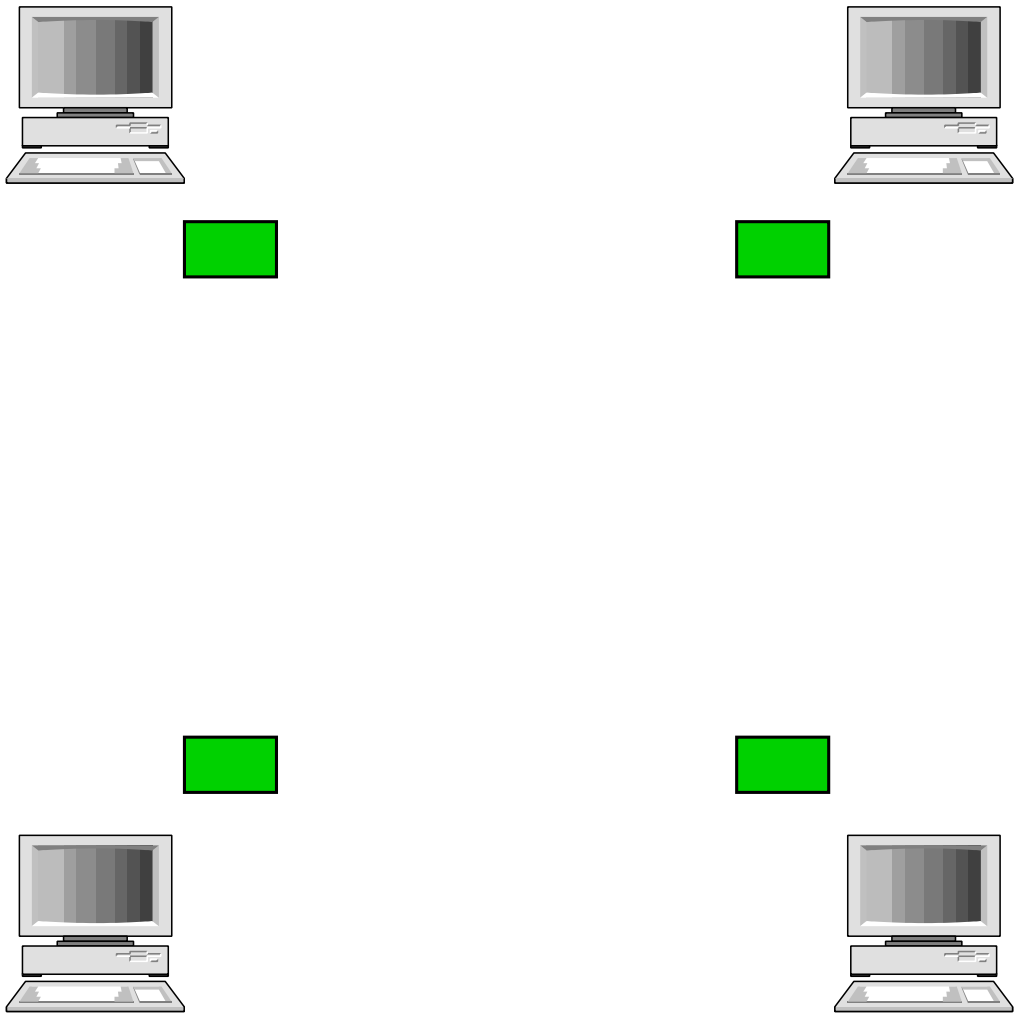
OTA: Změna struktury dat



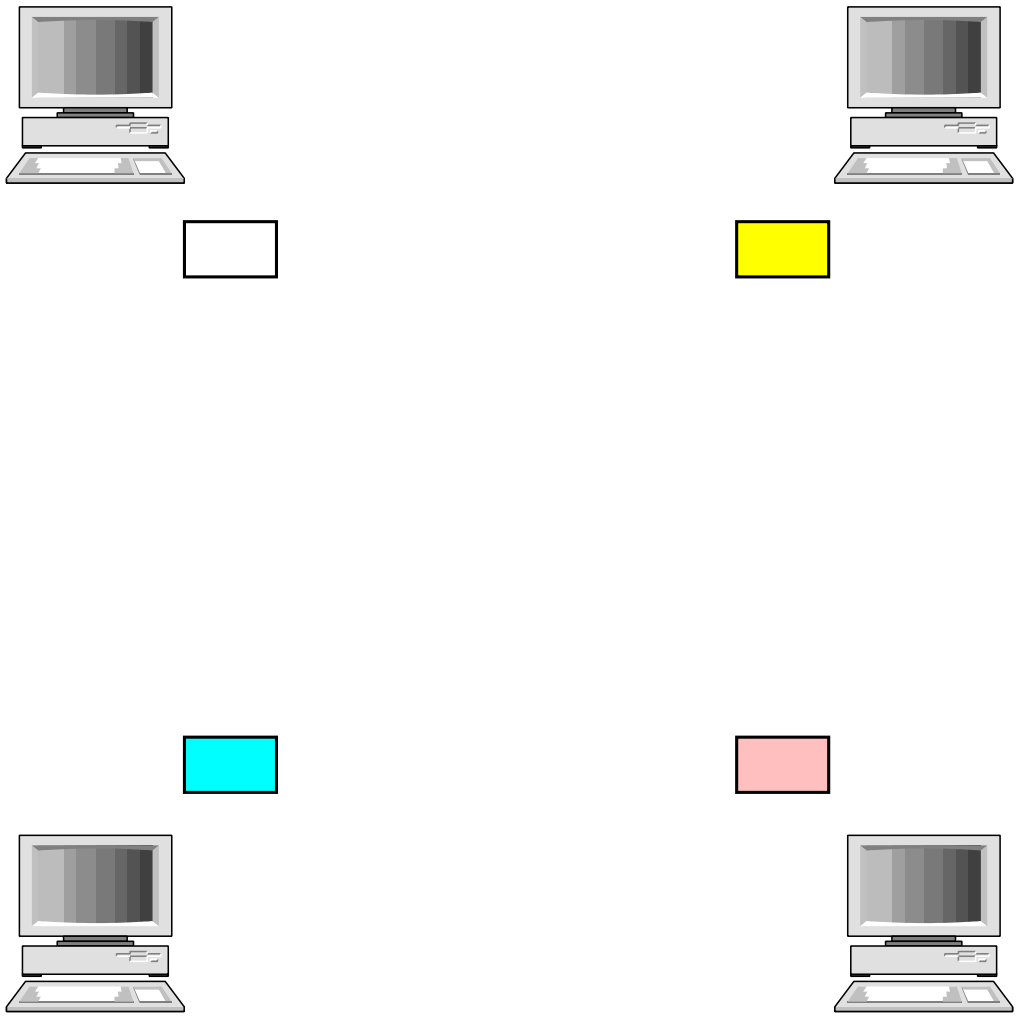
OTA: Změna struktury dat



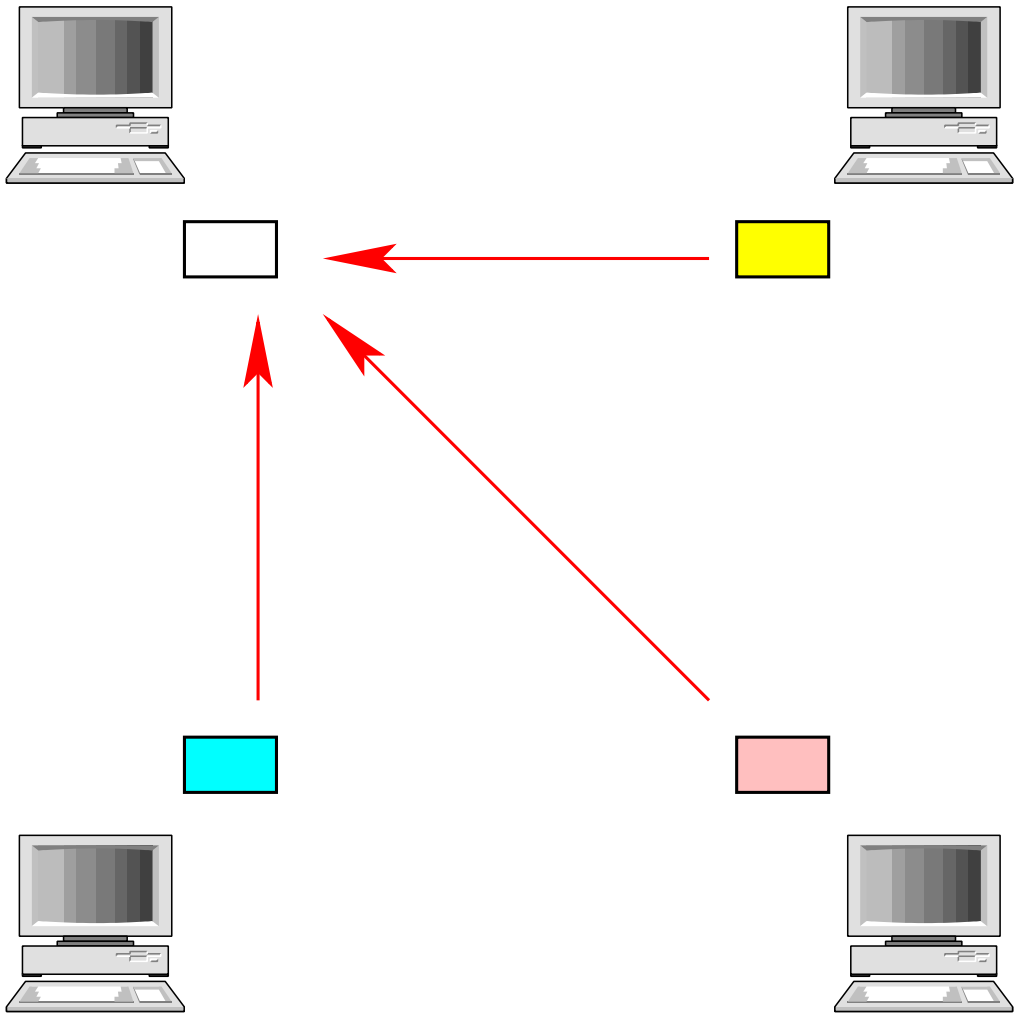
OTA: Změna struktury dat



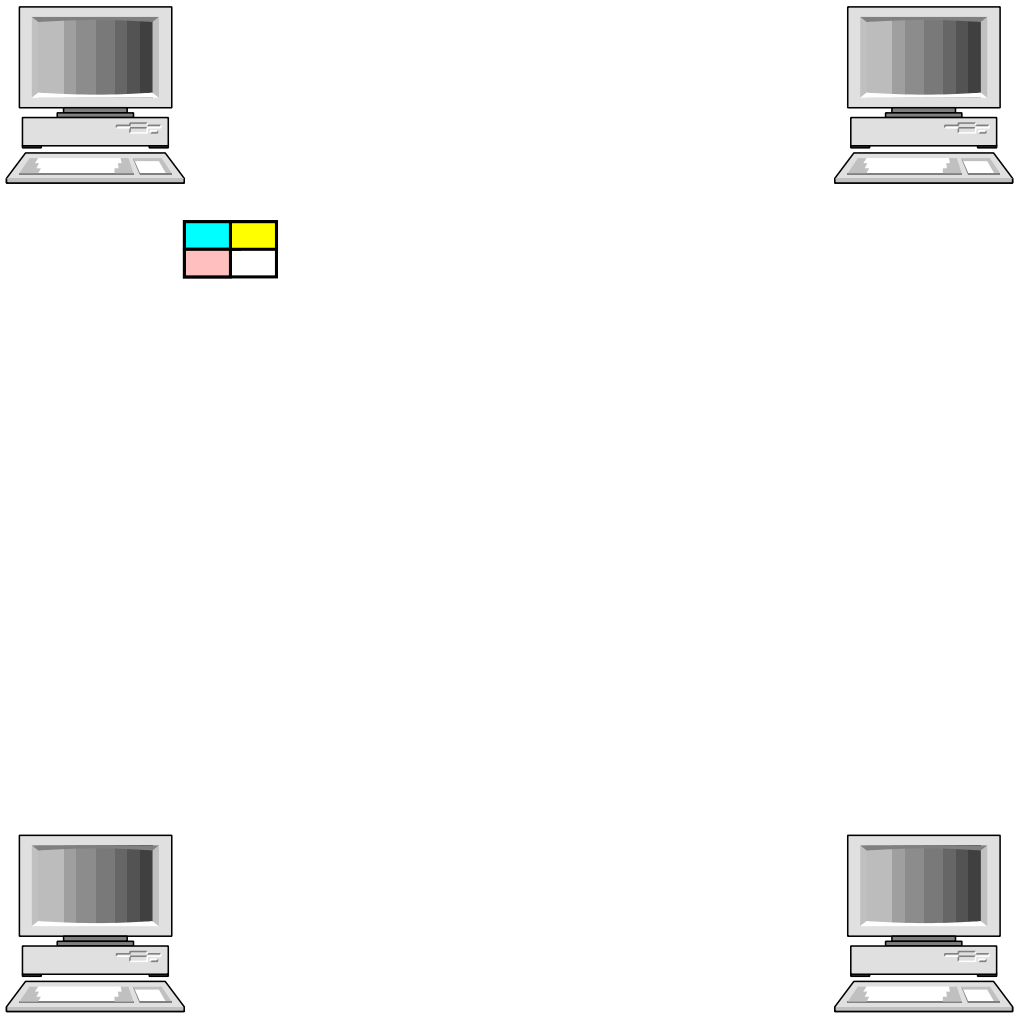
ATO: Změna struktury dat



ATO: Změna struktury dat



ATO: Změna struktury dat



OTA: Pro topologie prsten či řetěz

Naivní způsob One-To-All pro p procesů

- Poslat $p - 1$ zpráv postupně ostatním procesům
- Úzké místo: odesílatel
- Síť je nevyužitá, komunikuje pouze jedna dvojice procesů

Technika rekurzivního zdvojení (připomenutí)

- Nejprve první proces pošle zprávu jinému procesu
- Poté oba procesy pošlou zprávu jiné dvojici
- Poté čtveřice procesů pošle zprávu jiné čtveřici
- ...
- První proces pošle nejvýše $log(p)$ zpráv
- Souběh zpráv na linkách sítě
- Optimální vzdálenosti adresátů pro jednotlivá kola jsou $p/2$, $p/4$, $p/8$, $p/16$, ...

OTA: Pro topologii d-dimenzionální mřížka

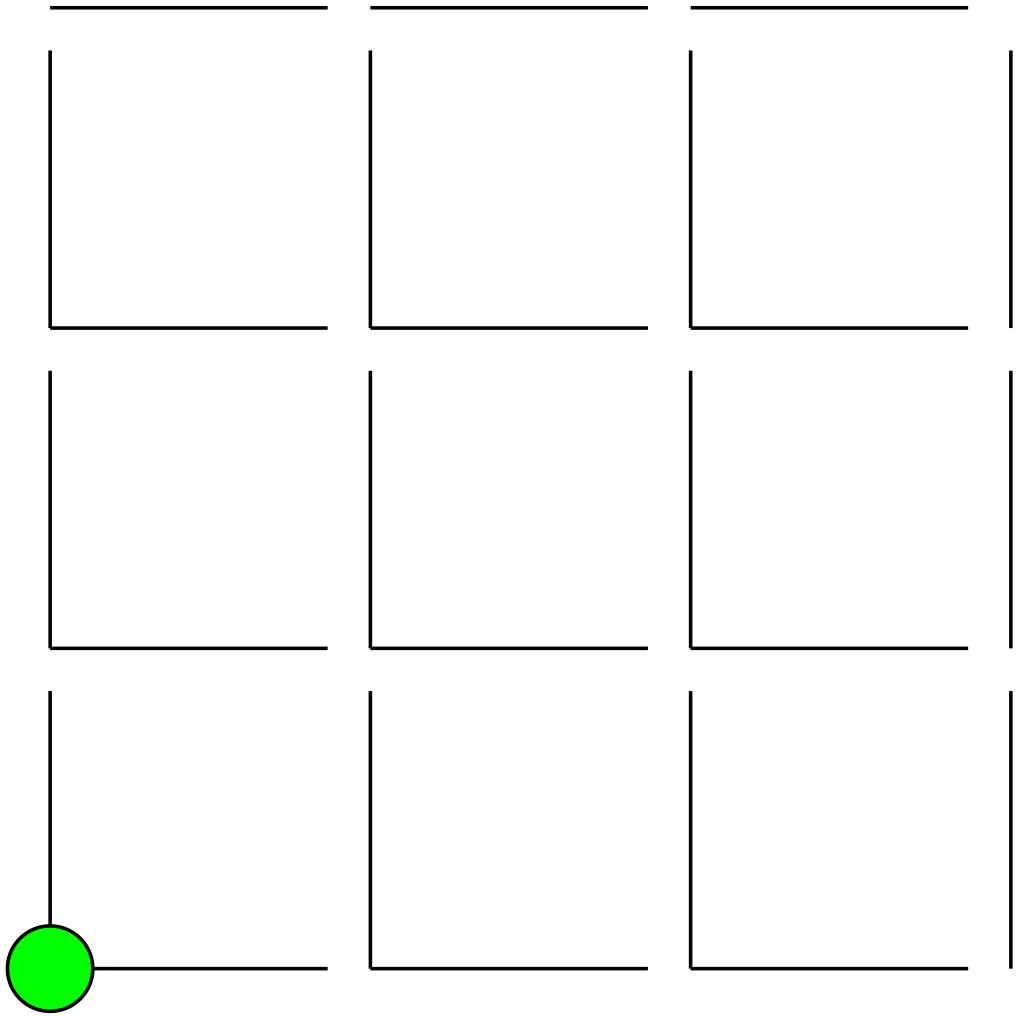
One-To-All ve 2-rozměrné síti o p uzlech

- Lze chápat jako $\sqrt{p}$ řetízků o $\sqrt{p}$ uzlech
- V první fázi se propaguje informace do všech řetízků
- V druhé fázi se souběžně propaguje informace v jednotlivých řetízcích
- Celková cena: $2(log(\sqrt{p}))$ sekvenčně provedených operací

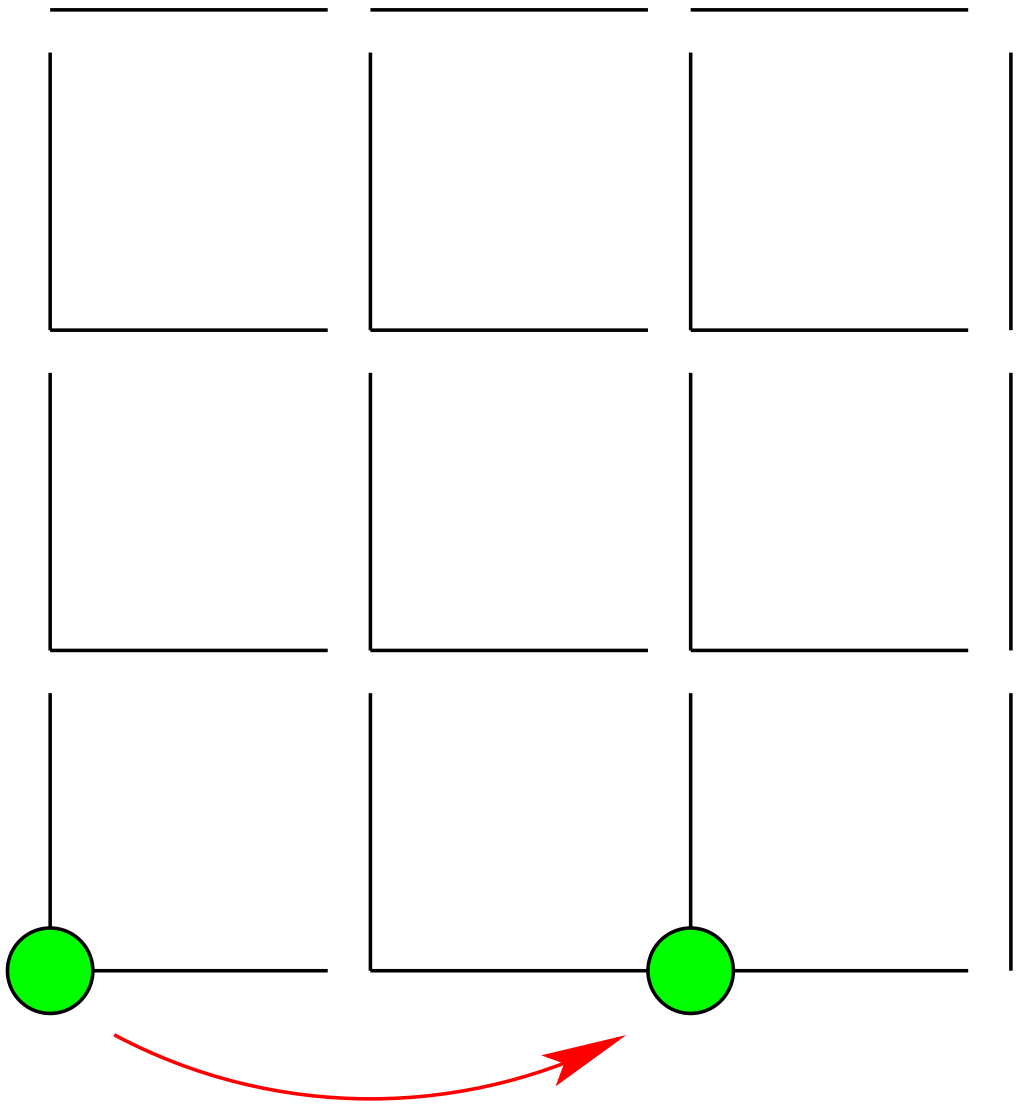
One-To-All v d-rozměrné síti

- Stejný princip, velikost v jednom rozměru je $p^{1/d}$
- Celková cena: $d(log(p^{1/d}))$

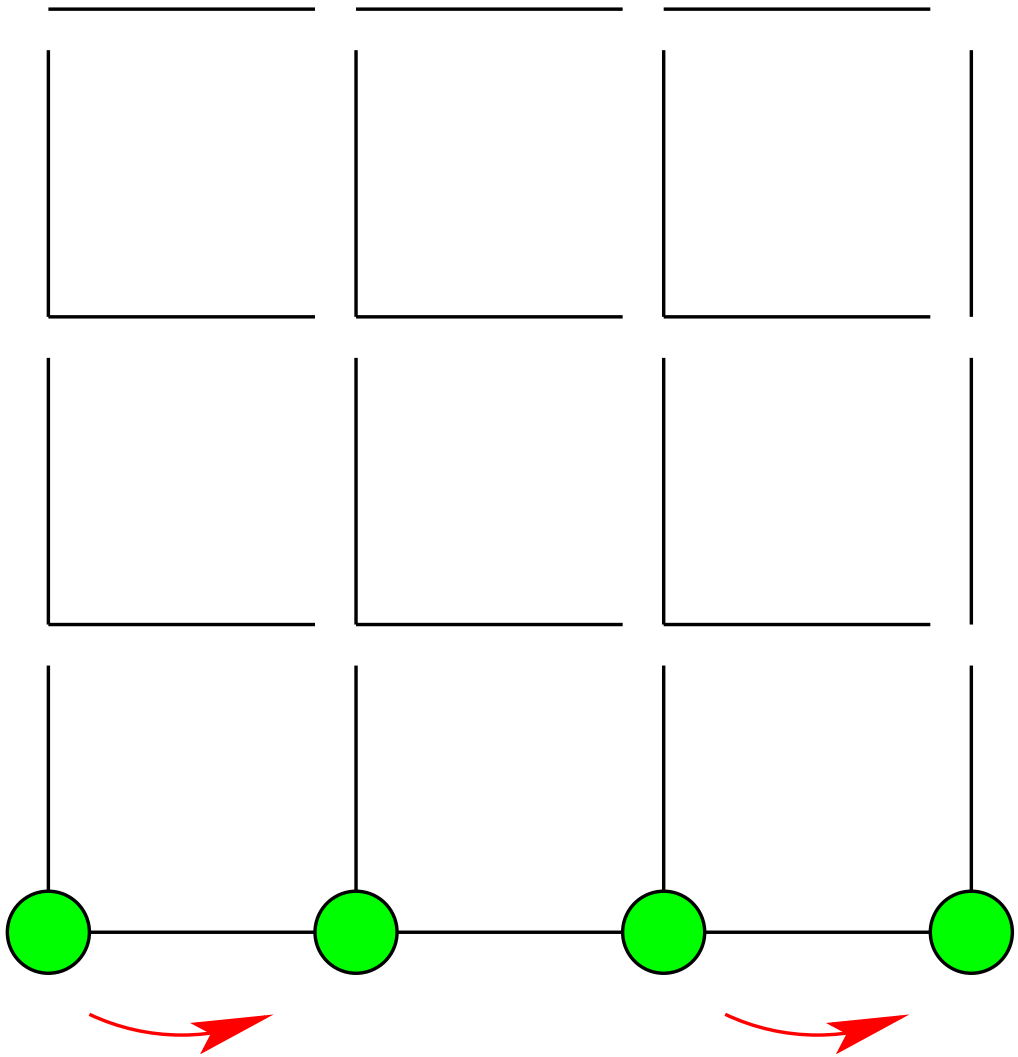
OTA: Pro topologii 2-dimenzionální mřížka



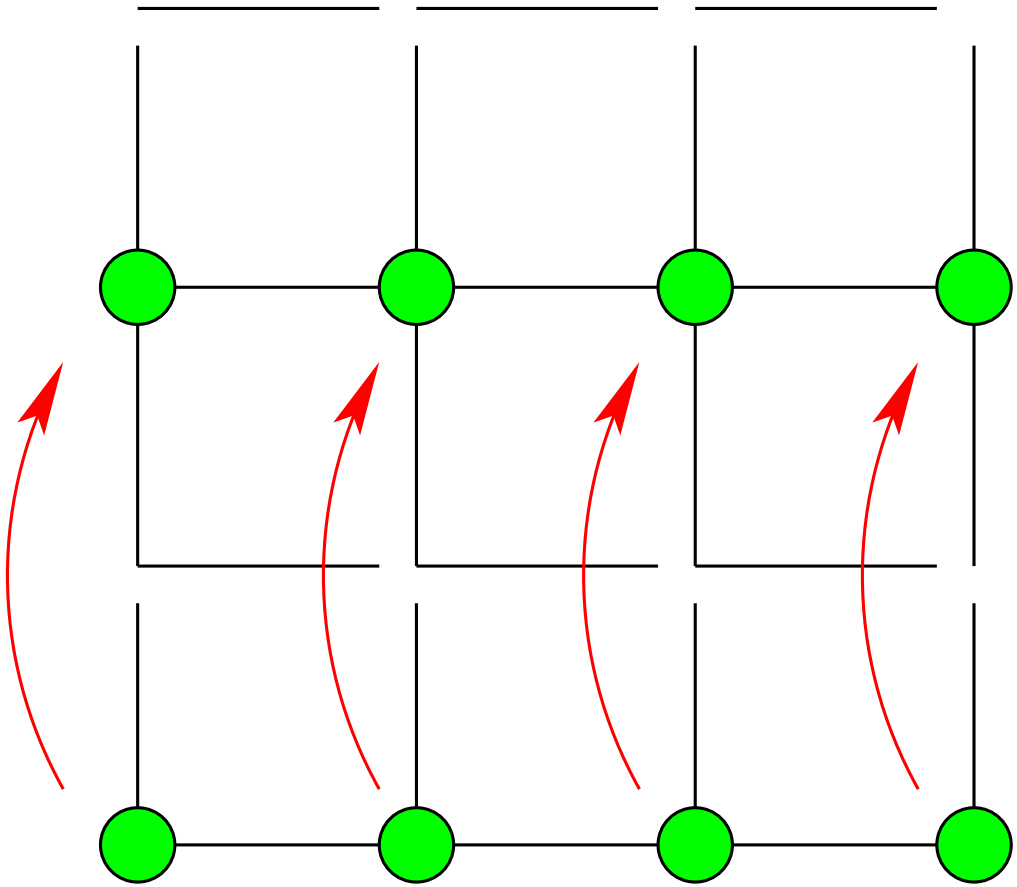
OTA: Pro topologii 2-dimenzionální mřížka



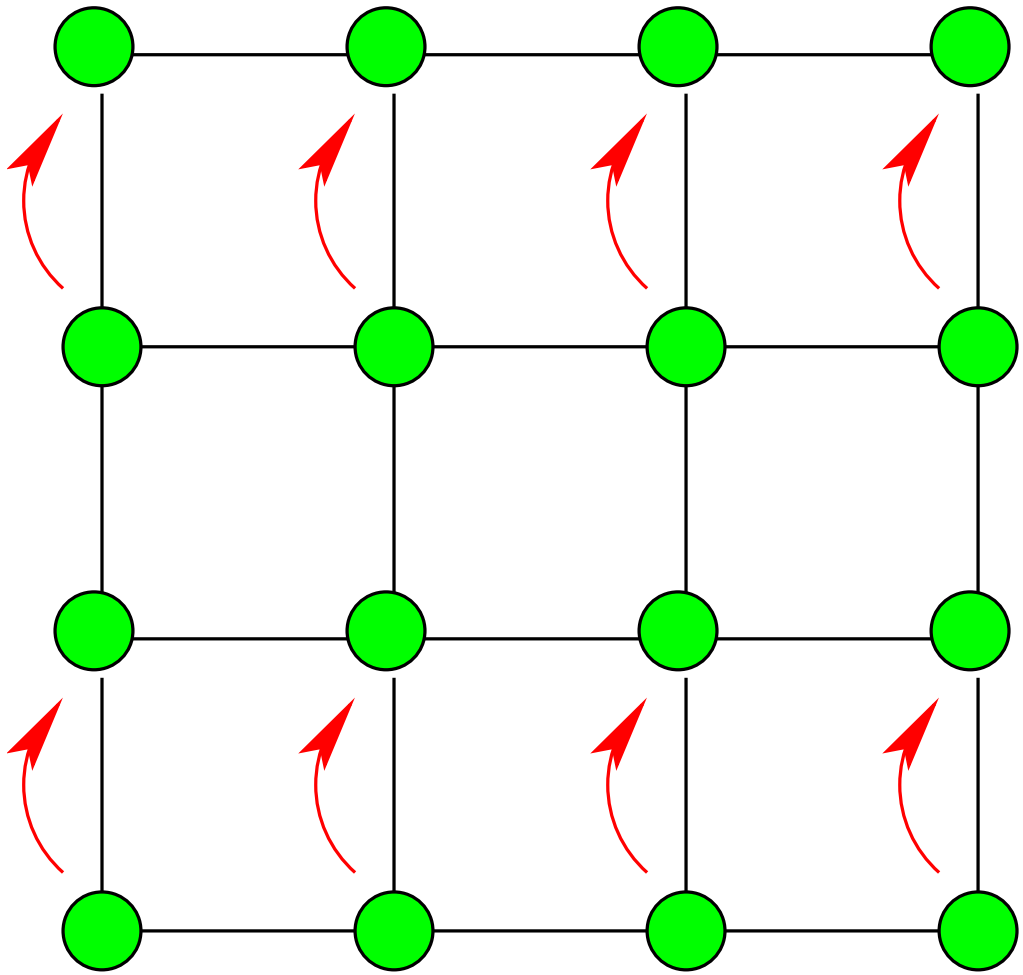
OTA: Pro topologii 2-dimenzionální mřížka



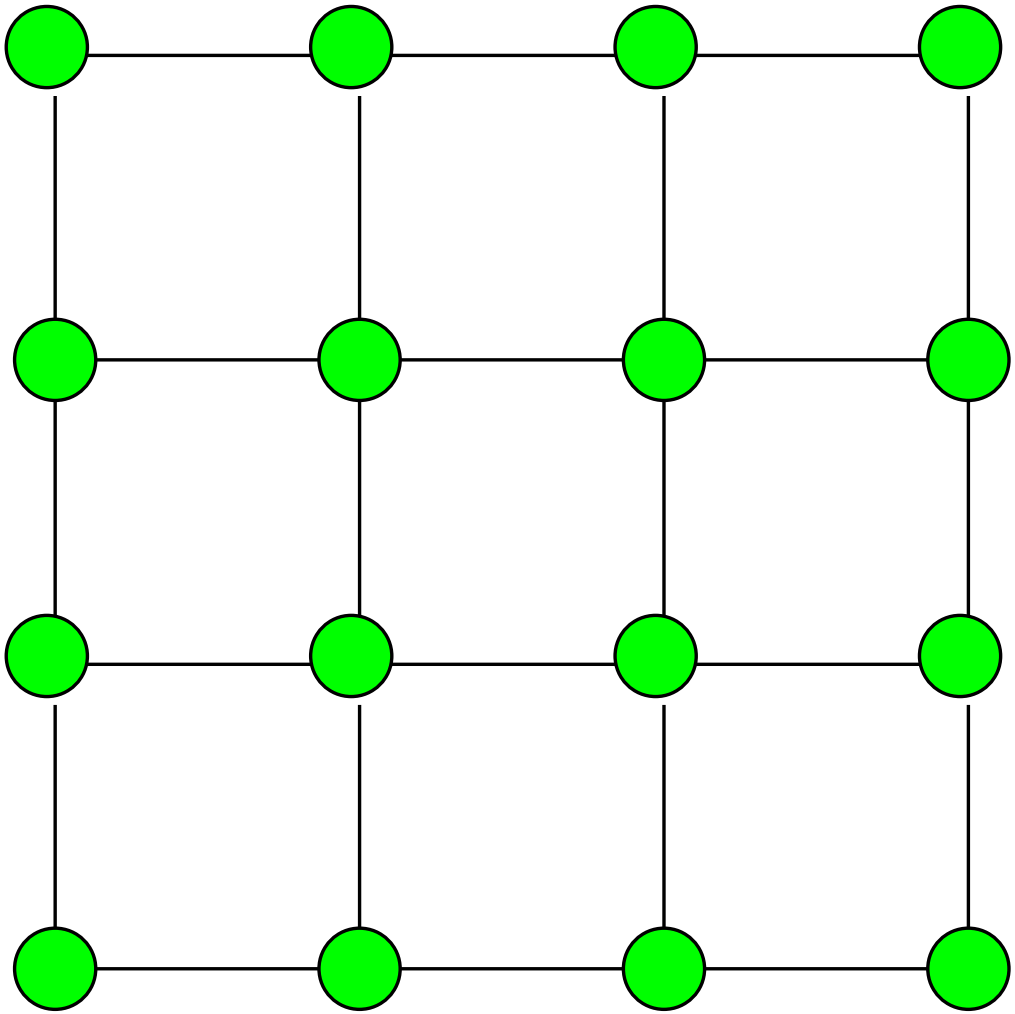
OTA: Pro topologii 2-dimenzionální mřížka



OTA: Pro topologii 2-dimenzionální mřížka



OTA: Pro topologii 2-dimenzionální mřížka



OTA: Pro topologii hyperkostka

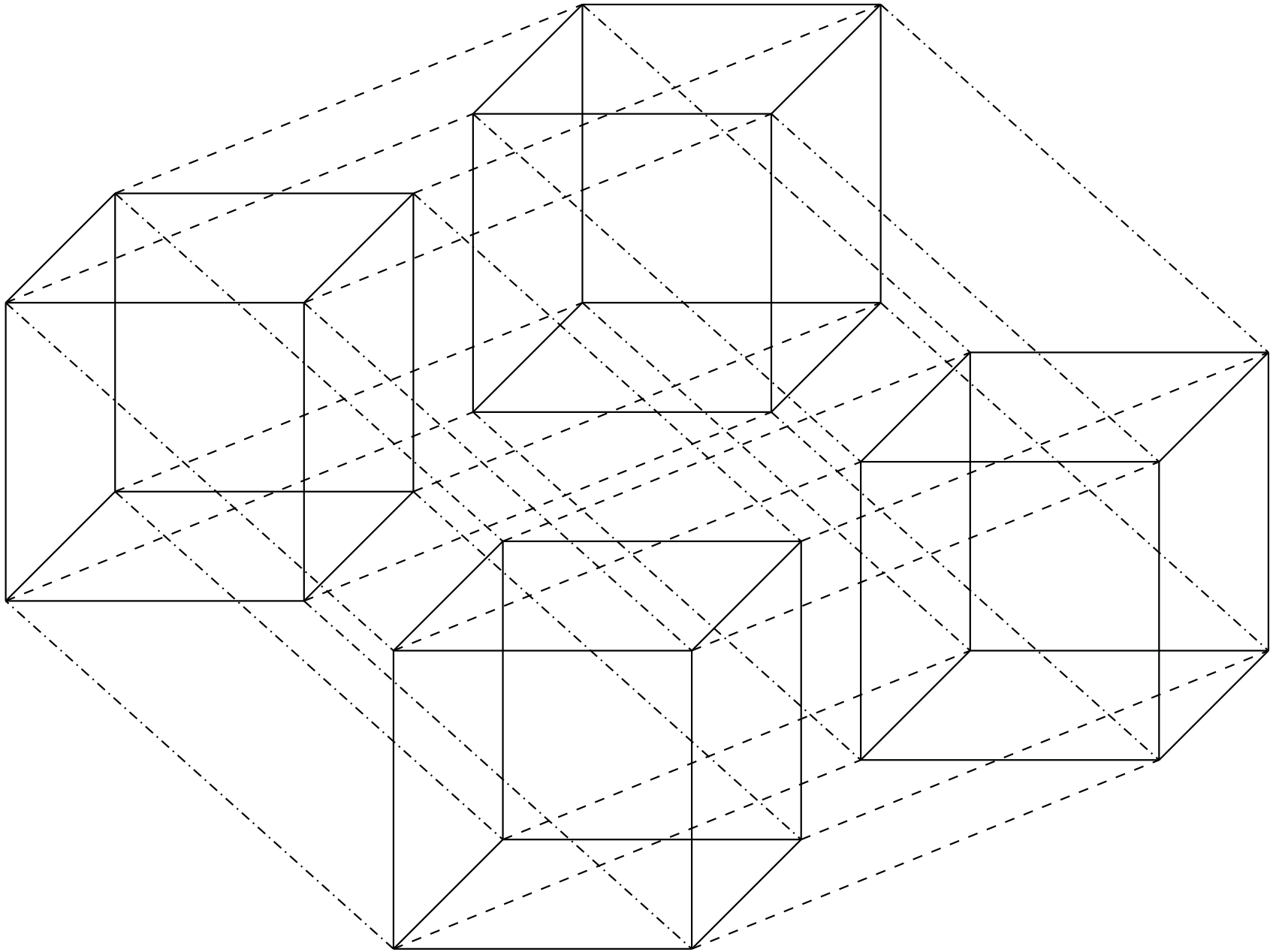
Hyperkostka s 2^d vrcholy

- Aplikuje se algoritmus pro síť ve tvaru mřížky
- d-dimenzionální síť s hloubkou sítě 2
- Každá fáze proběhne v konstantním čase (poslání 1 zprávy)
- Nenastává souběžný přístup na žádnou z komunikačních linek
- Celková cena: d

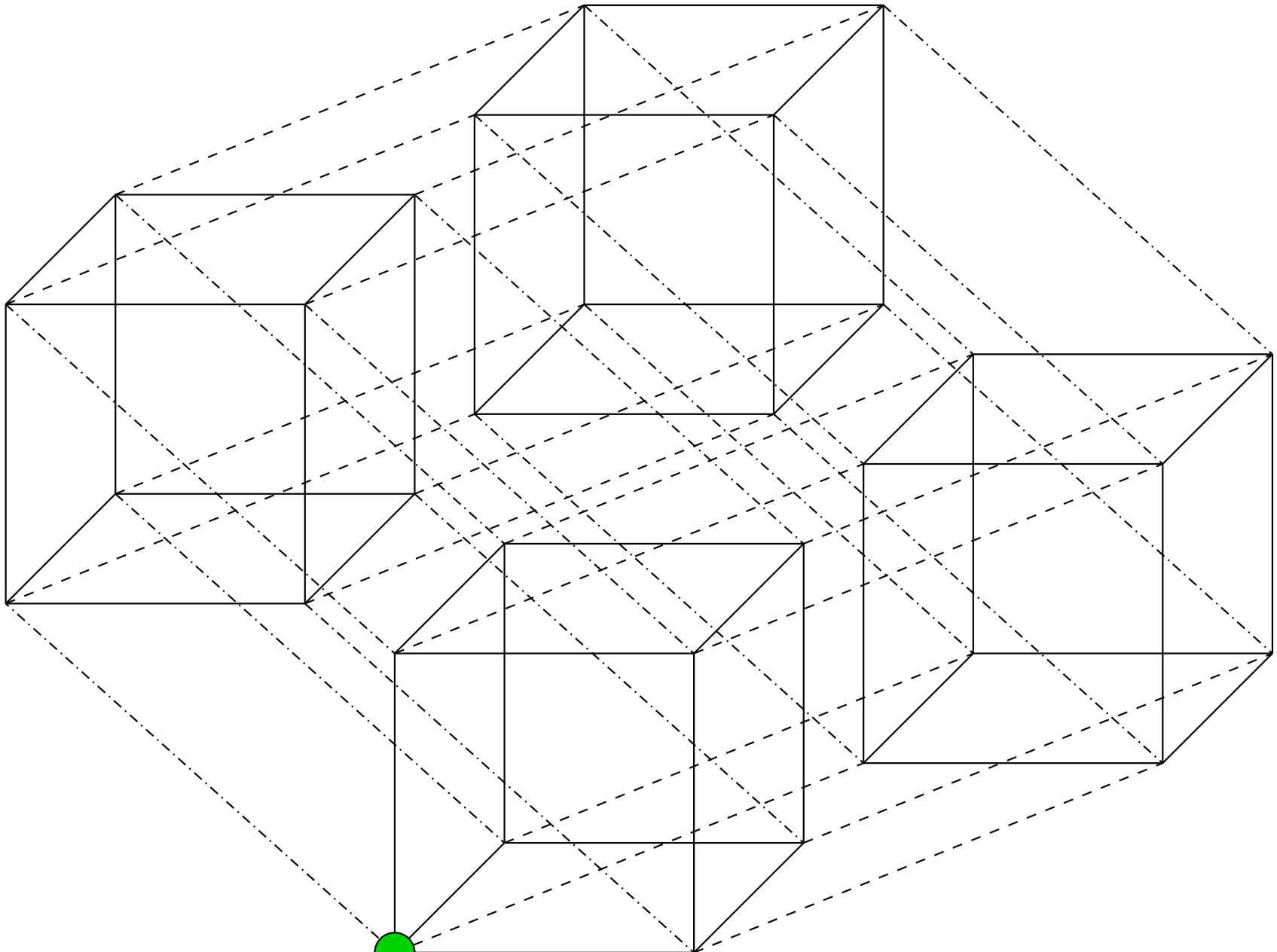
Příklad

- Jak proběhne One-To-All na hyperkostce s dimenzí 5?

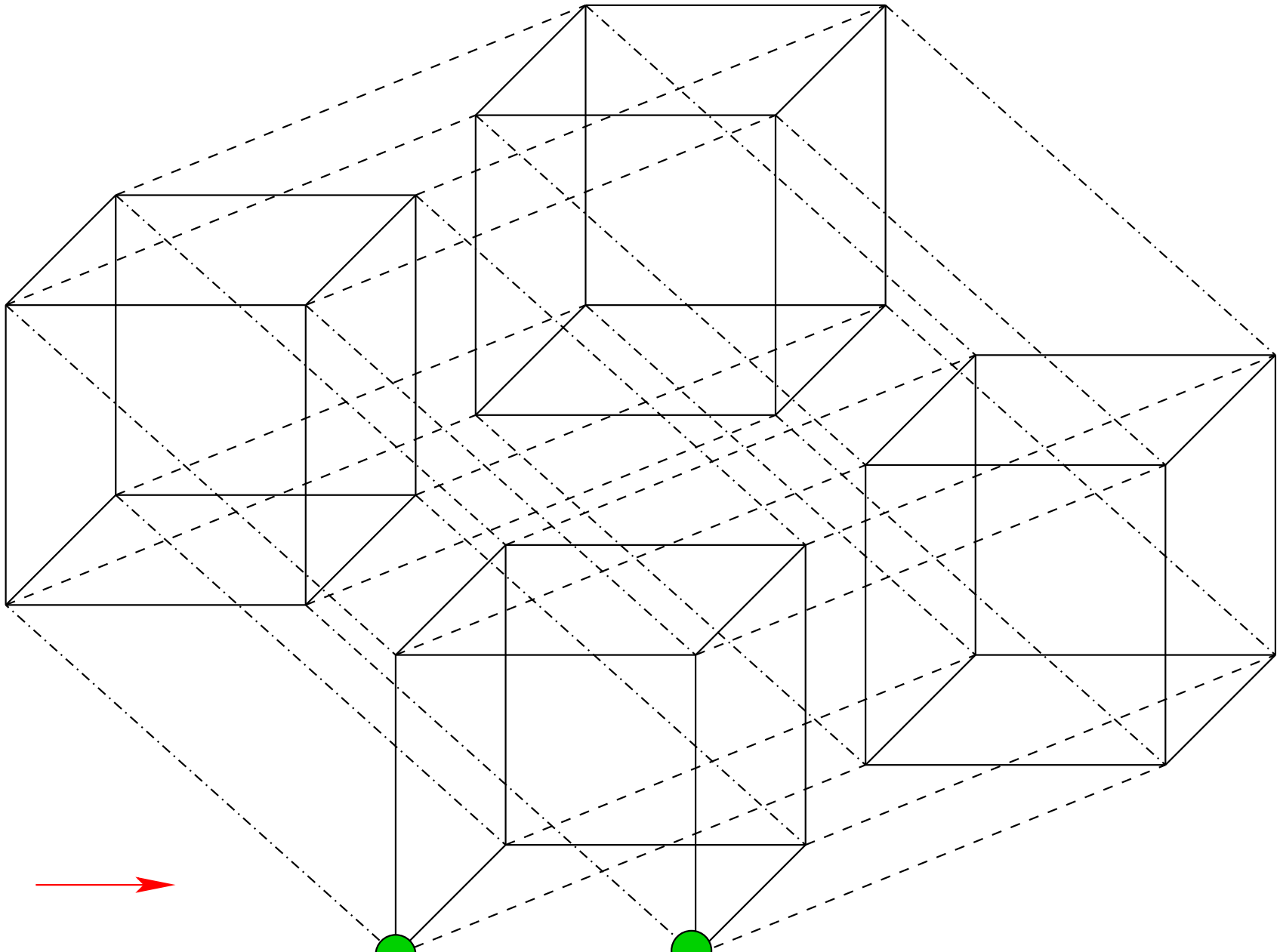
OTA: Pro hyperkostku o dimenzi 5



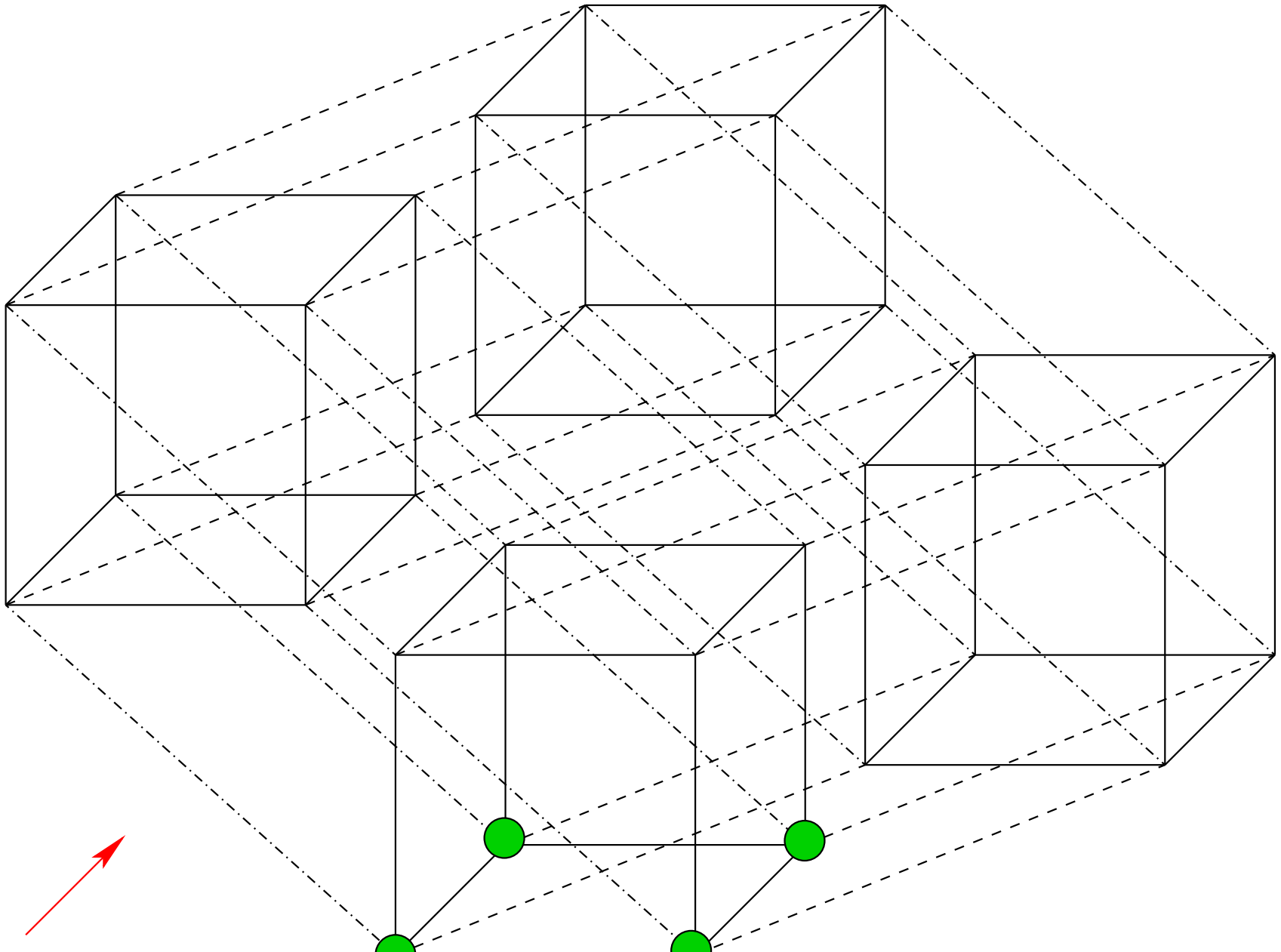
OTA: Pro hyperkostku o dimenzi 5



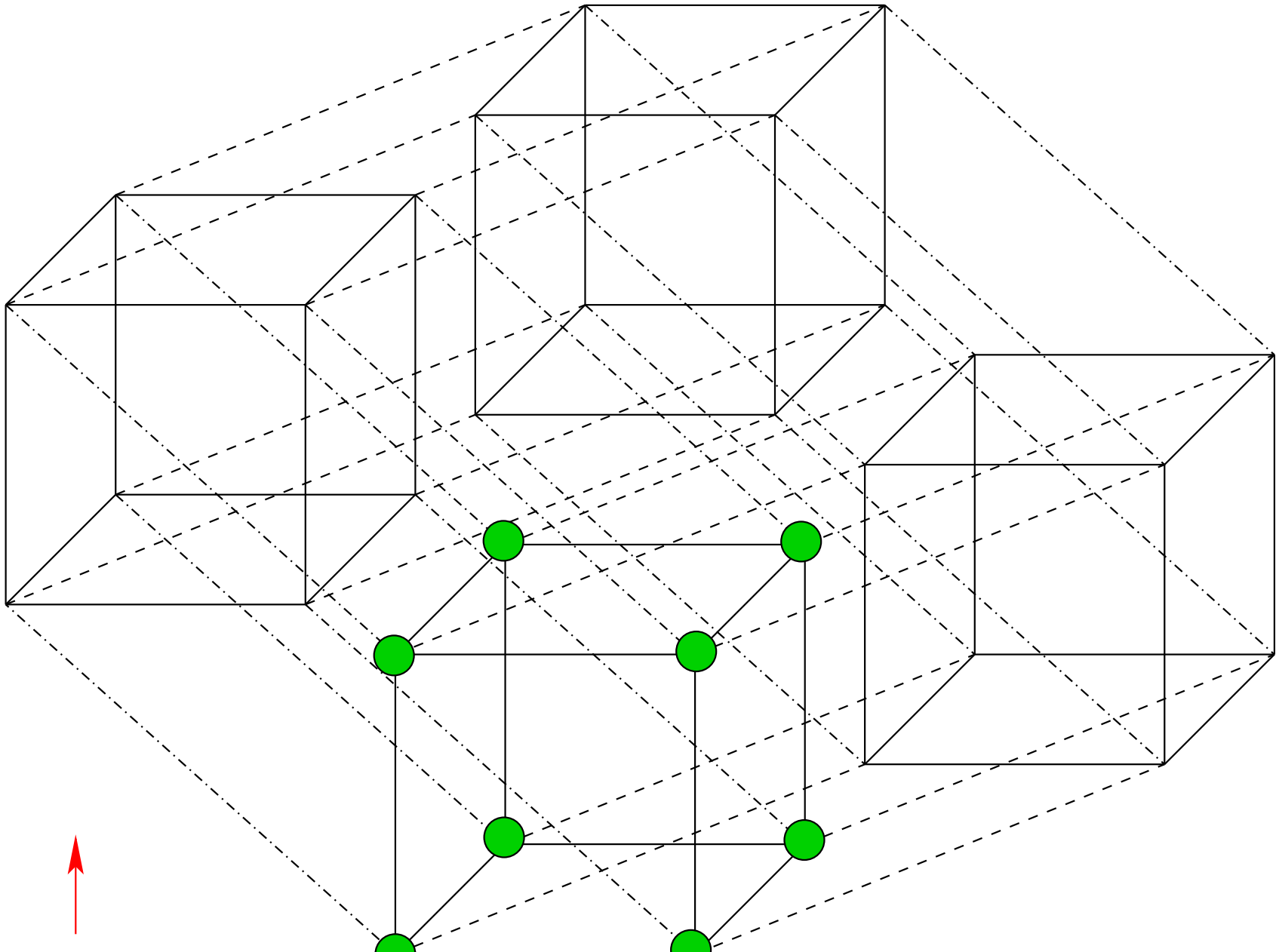
OTA: Pro hyperkostku o dimenzi 5



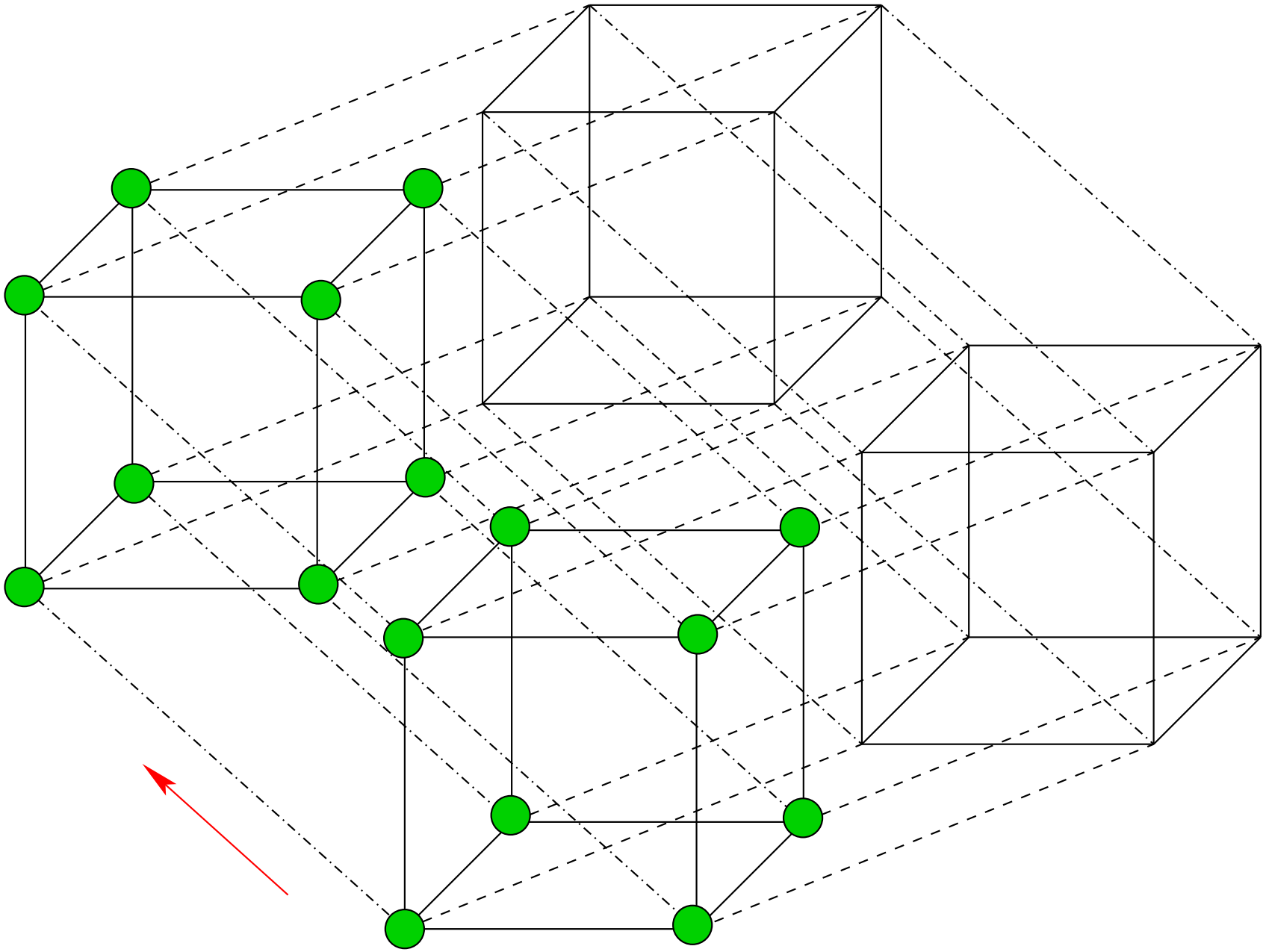
OTA: Pro hyperkostku o dimenzi 5



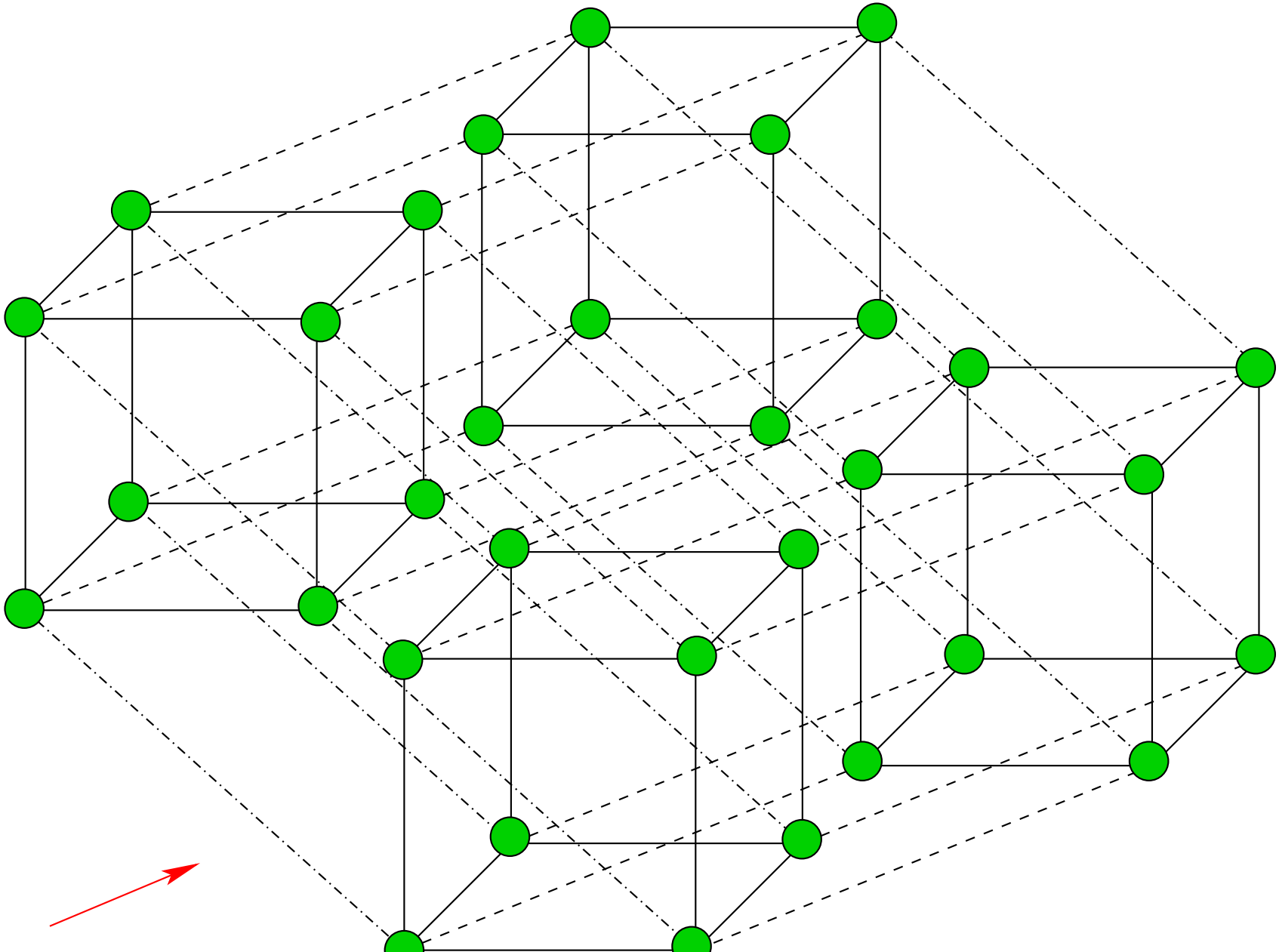
OTA: Pro hyperkostku o dimenzi 5



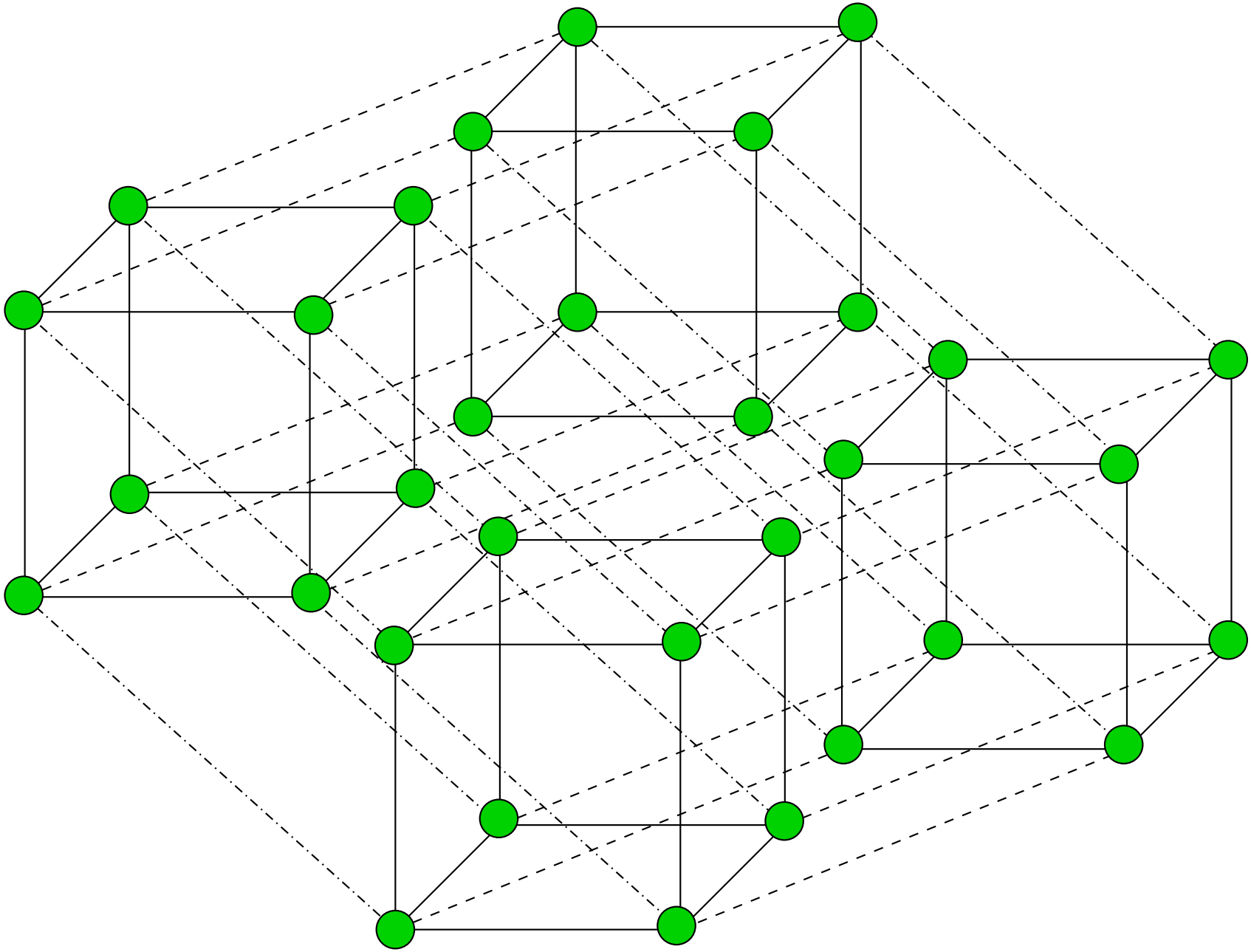
OTA: Pro hyperkostku o dimenzi 5



OTA: Pro hyperkostku o dimenzi 5



OTA: Pro hyperkostku o dimenzi 5



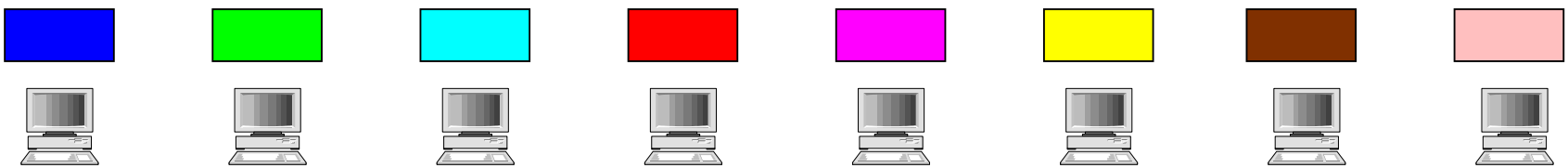
ATO: Duálně k OTA

All-To-One Redukce pro p procesů

- Probíhá duálně k operaci One-To-All

Příklad: ATO pro topologie prsten či řetěz

- Prvně procesy s lichým ID pošlou zprávu procesům s ID o jedna menším, kde se zprávy zkombinují s hodnotou, kterou chtějí vyslat procesy se sudým ID
- Následně proběhne kombinace informací sousedních procesů se sudým ID na procesech jejichž ID je násobkem čtyř
- ...



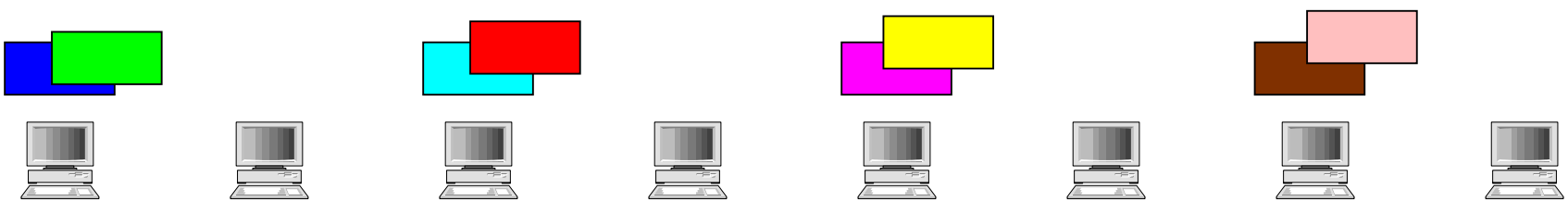
ATO: Duálně k OTA

All-To-One Redukce pro p procesů

- Probíhá duálně k operaci One-To-All

Příklad: ATO pro topologie prsten či řetěz

- Prvně procesy s lichým ID pošlou zprávu procesům s ID o jedna menším, kde se zprávy zkombinují s hodnotou, kterou chtějí vyslat procesy se sudým ID
- Následně proběhne kombinace informací sousedních procesů se sudým ID na procesech jejichž ID je násobkem čtyř
- ...



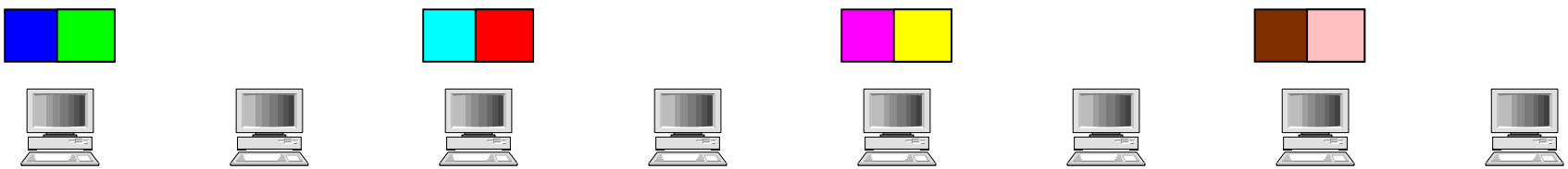
ATO: Duálně k OTA

All-To-One Redukce pro p procesů

- Probíhá duálně k operaci One-To-All

Příklad: ATO pro topologie prsten či řetěz

- Prvně procesy s lichým ID pošlou zprávu procesům s ID o jedna menším, kde se zprávy zkombinují s hodnotou, kterou chtějí vyslat procesy se sudým ID
- Následně proběhne kombinace informací sousedních procesů se sudým ID na procesech jejichž ID je násobkem čtyř
- ...



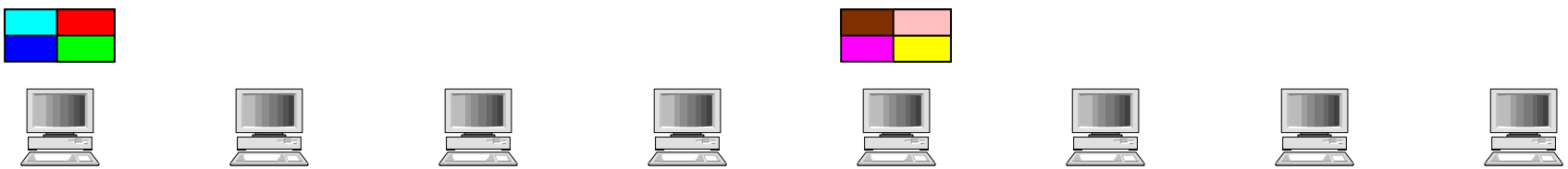
ATO: Duálně k OTA

All-To-One Redukce pro p procesů

- Probíhá duálně k operaci One-To-All

Příklad: ATO pro topologie prsten či řetěz

- Prvně procesy s lichým ID pošlou zprávu procesům s ID o jedna menším, kde se zprávy zkombinují s hodnotou, kterou chtějí vyslat procesy se sudým ID
- Následně proběhne kombinace informací sousedních procesů se sudým ID na procesech jejichž ID je násobkem čtyř
- ...



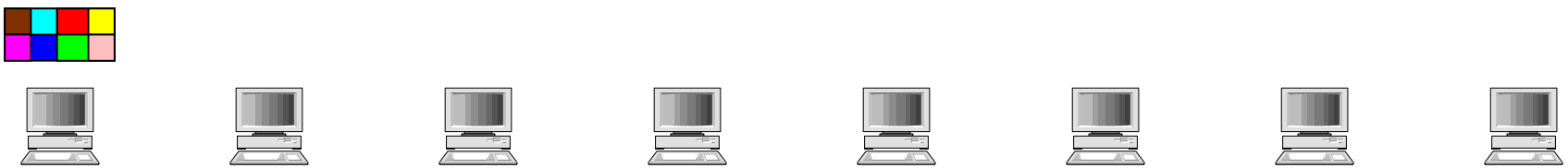
ATO: Duálně k OTA

All-To-One Redukce pro p procesů

- Probíhá duálně k operaci One-To-All

Příklad: ATO pro topologie prsten či řetěz

- Prvně procesy s lichým ID pošlou zprávu procesům s ID o jedna menším, kde se zprávy zkombinují s hodnotou, kterou chtějí vyslat procesy se sudým ID
- Následně proběhne kombinace informací sousedních procesů se sudým ID na procesech jejichž ID je násobkem čtyř
- ...



Univerzální algoritmy

Pozorování

- One-To-All procedury jsou si podobné
- Jdou nahradit univerzální procedurou

Univerzální algoritmy OTA vysílání a ATO Redukce

- Předpokládají 2^d uzlů (hyperkostka)
- Každý uzel identifikován bitovým vektorem
- *AND* a *XOR* bitové operace

Cena

- $d(t_s + mt_w)$

Univerzální algoritmy – OTA vysílání

```
(1)  proc One-To-All(d, id, X)
(2)     mask := 2^d − 1
(3)     for i := d − 1 downto 0
(4)        mask := mask XOR 2^i
(5)        if (id AND mask) = 0
(6)           then if (id AND 2^i) = 0
(7)                    then msg_destination := id XOR 2^i
(8)                         send X to msg_destination
(9)                    else msg_source := id XOR 2^i
(10)                        receive X from msg_source
(11)                fi
(12)       fi
(13)    end
(14) end
```

Univerzální algoritmy – OTA vysílání – libovolný odesílatel

```
(1)  proc General-One-To-All(d, id, src, X)
(2)      virtid := id XOR src
(3)      mask := 2^d − 1
(4)      for i := d − 1 downto 0
(5)          mask := mask XOR 2^i
(6)          if (virtid AND mask) = 0
(7)            then if (virtid AND 2^i) = 0
(8)                    then virt_destination := virtid XOR 2^i
(9)                         send X to (virt_destination XOR src)
(10)                  else virt_source := virtid XOR 2^i
(11)                        receive X from (virt_source XOR src)
(12)                 fi
(13)         fi
(14)     end
(15) end
```

Univerzální algoritmy – ATO Redukce

```
(1)  proc ALL-TO-ONE(d, id, m, X, sum)
(2)     for j := 0 to m − 1 do sum[j] := X[j] end
(3)     mask := 0
(4)     for i := to d − 1
(5)         if (id AND mask) = 0
(6)           then if (id AND 2^i) = 0
(7)                   then msg_destination := id XOR 2^i
(8)                        send sum to msg_destination
(9)                   else msg_source := id XOR 2^i
(10)                       receive X from msg_source
(11)                       for j := 0 to m − 1
(12)                           sum[j] := sum[j] + X[j]
(13)                       end
(14)                fi fi
(15)        mask := mask XOR 2^i
(16)    end
(17) end
```

Všichni všem a všichni od všech

Všichni všem a všichni od všech

All-To-All Vysílání

- Všichni posílají informaci všem
- Každá úloha posílá jednu zprávu všem ostatním úlohám
- Ve výsledku je p kopií p originálních zpráv v lokálních adresových prostorech adresátů
- Změna struktury dat: $p * m \mapsto p * p * m$

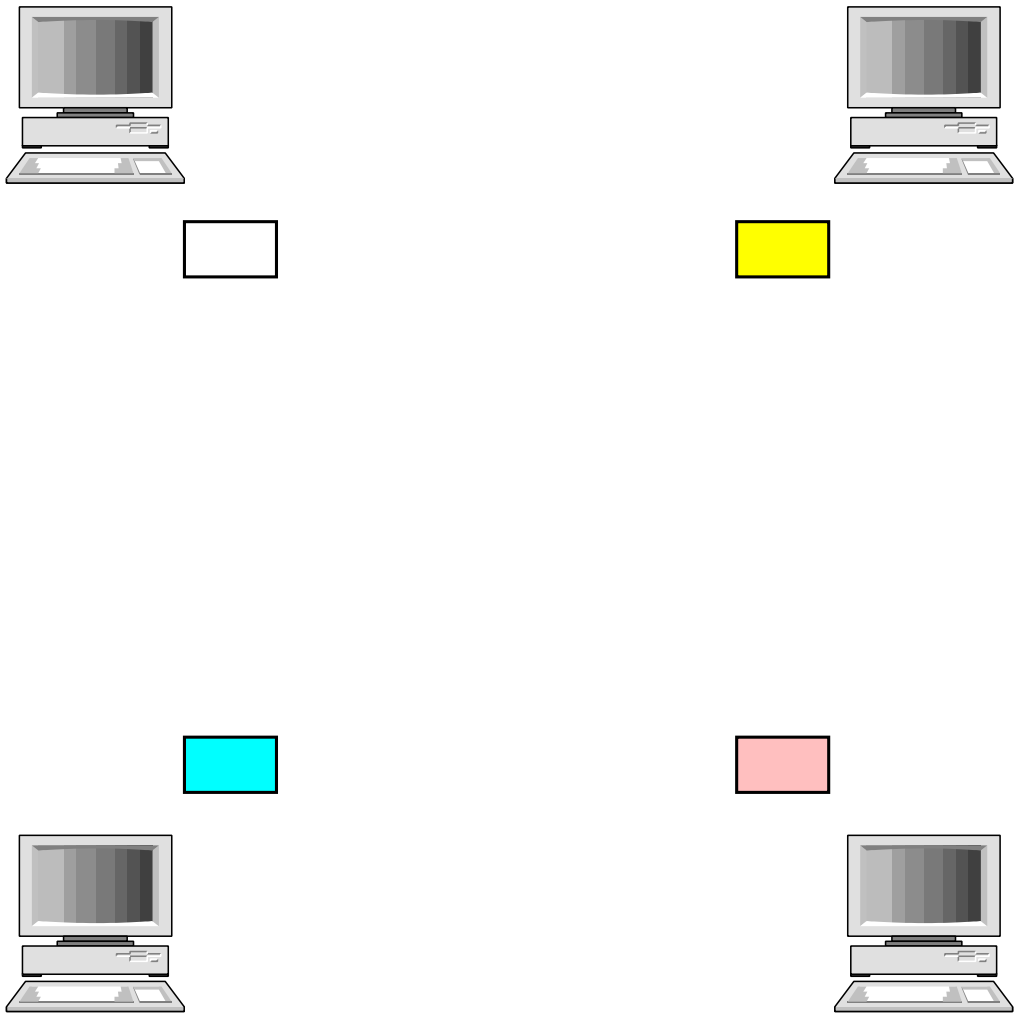
All-To-All Redukce

- Všichni posílají informaci všem, příchozí zprávy se kombinují
- Každá úloha má pro každou jinou úlohu jinou zprávu
- Změna struktury dat: $p * p * m \mapsto p * m$

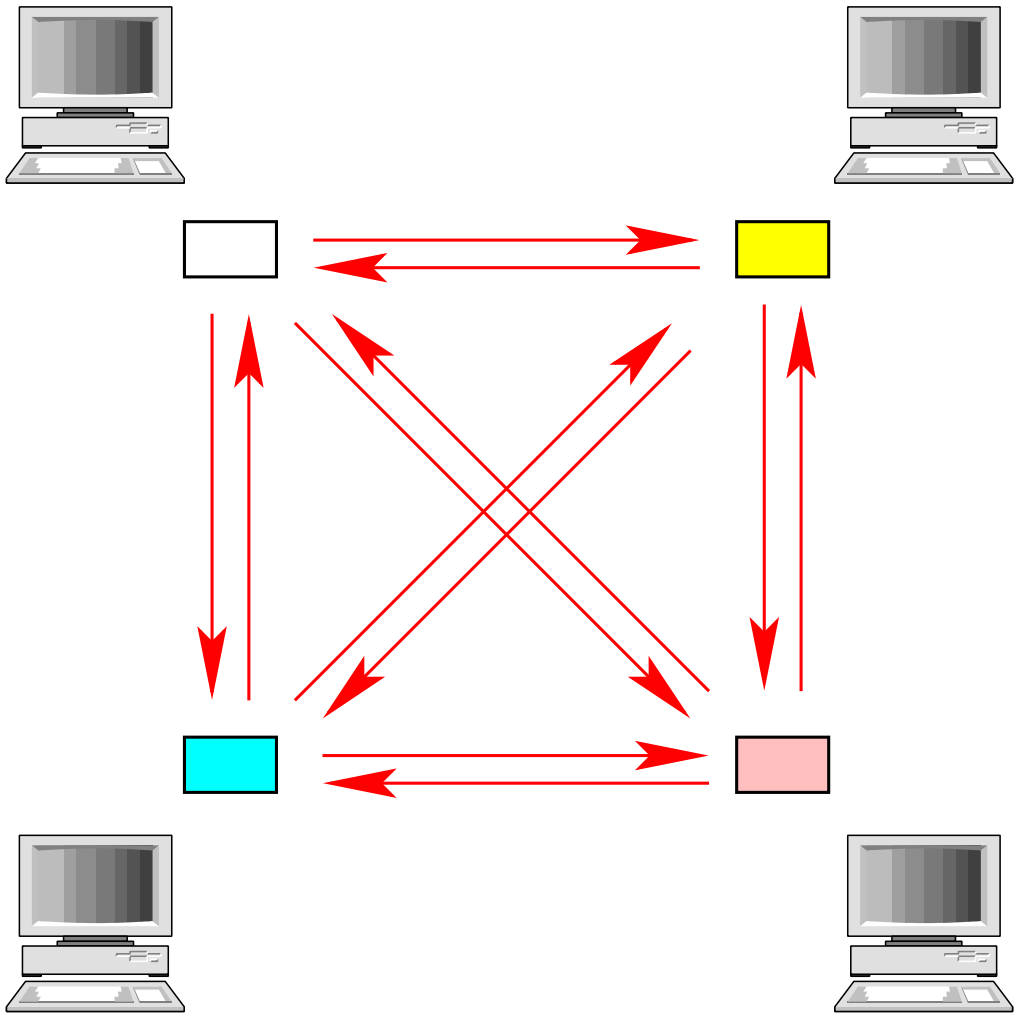
Naivní řešení

- Sekvenční/násobné použití One-To-All procedur
- Neefektivní využití sítě

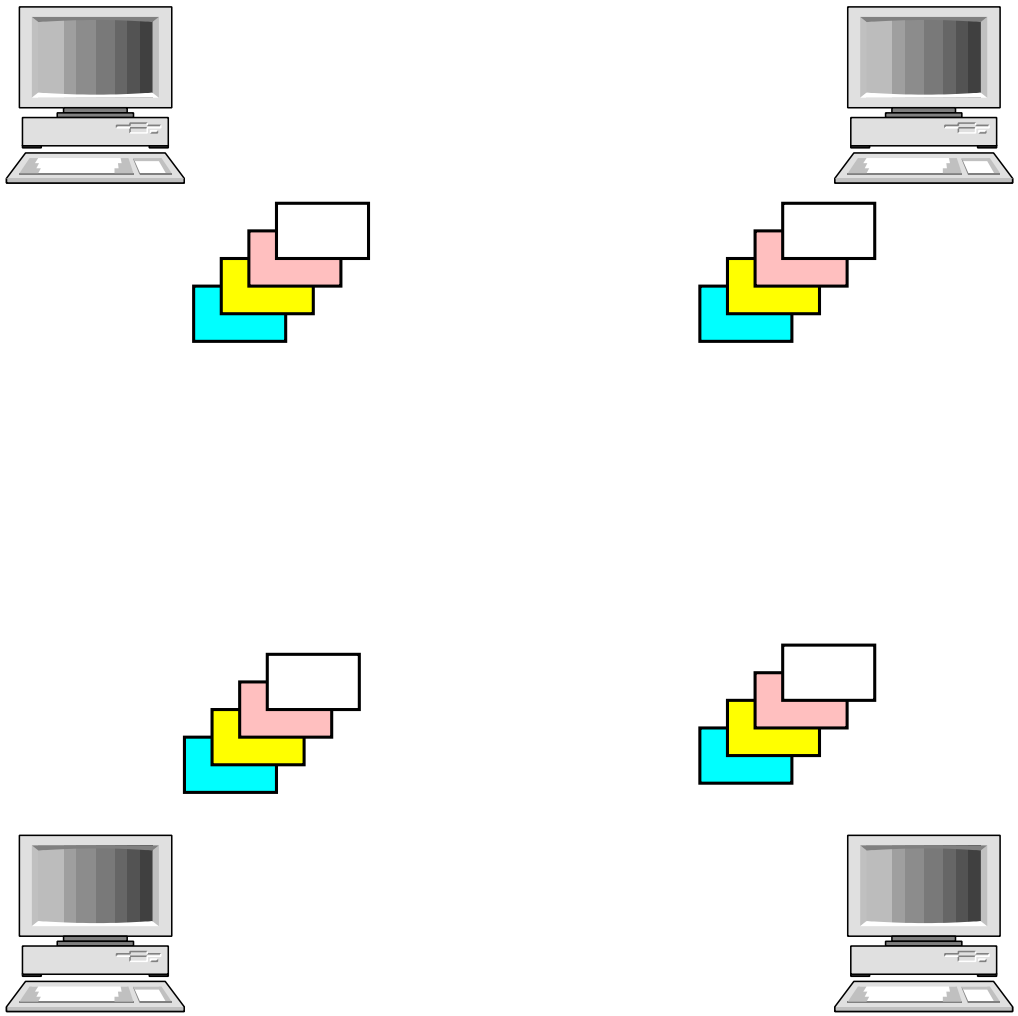
ATA Vysílání: Změna struktury dat



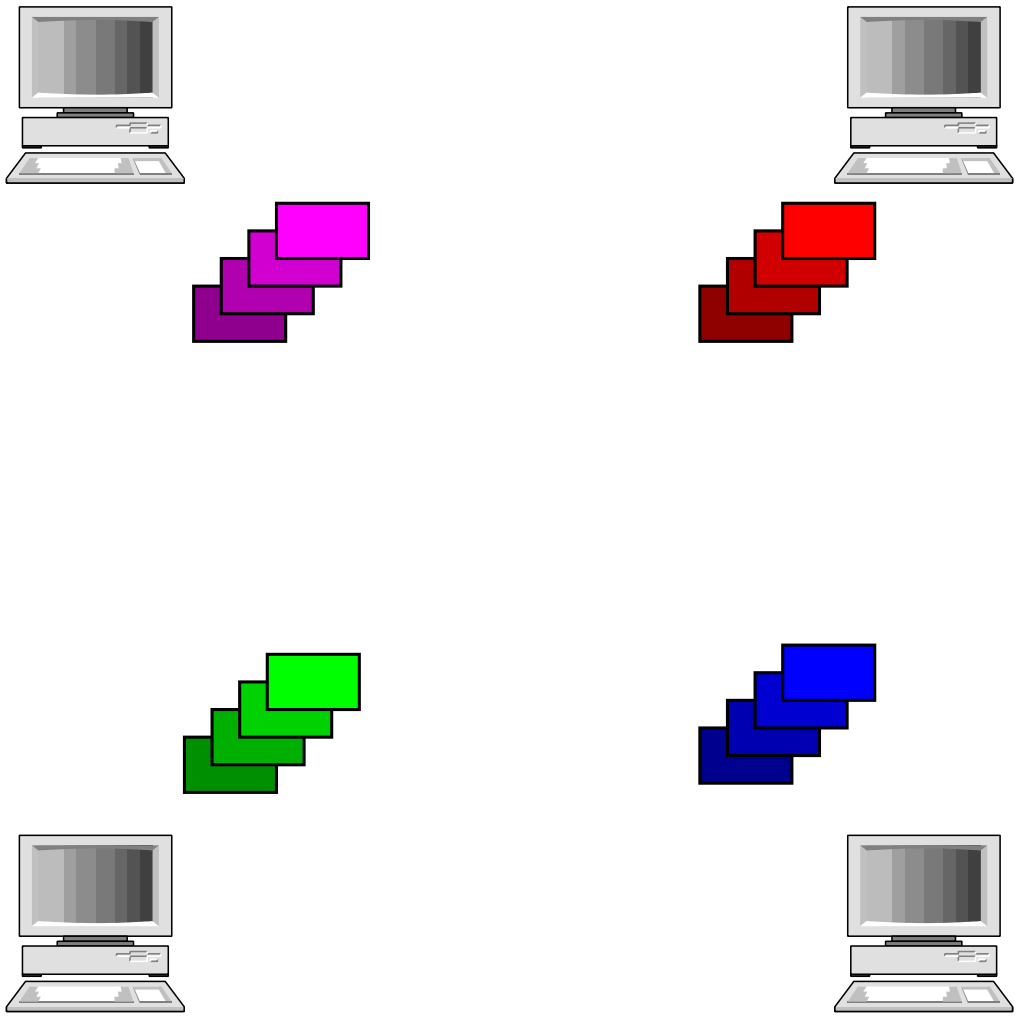
ATA Vysílání: Změna struktury dat



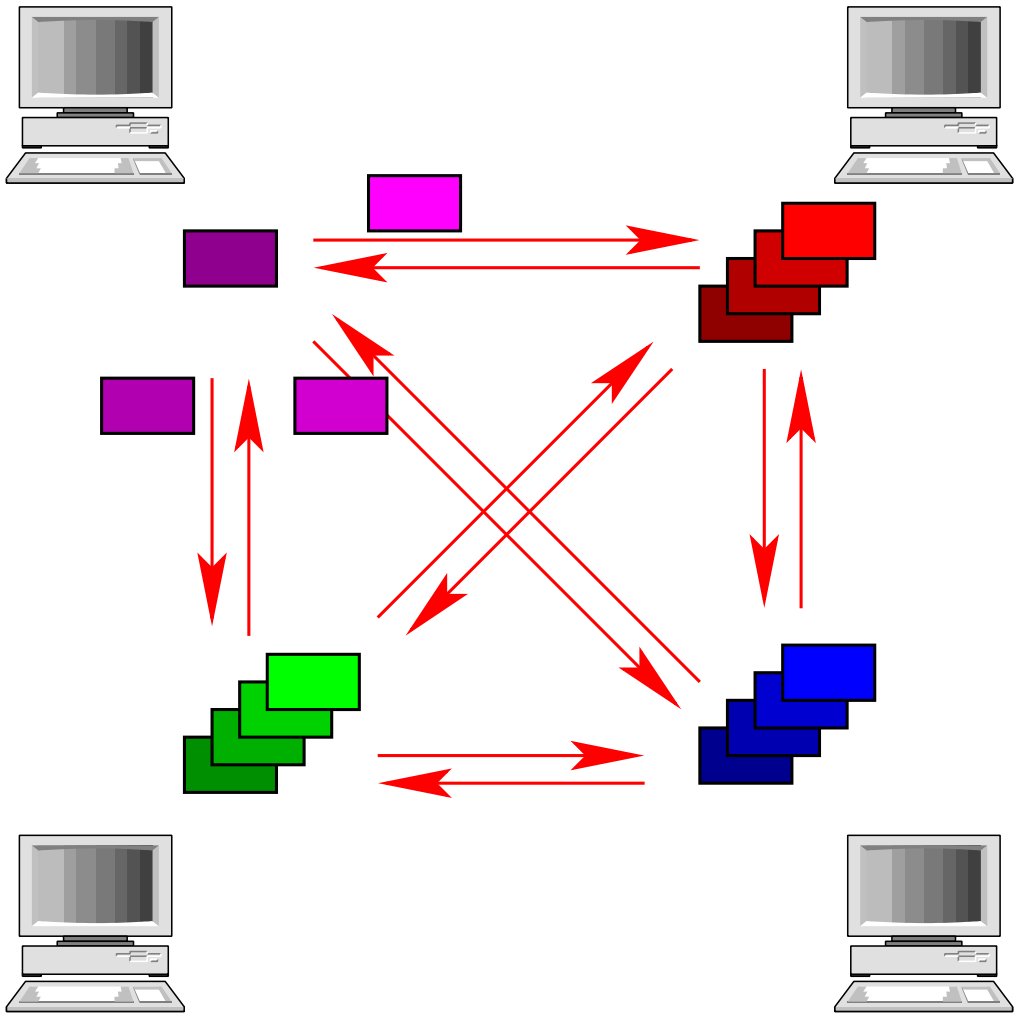
ATA Vysílání: Změna struktury dat



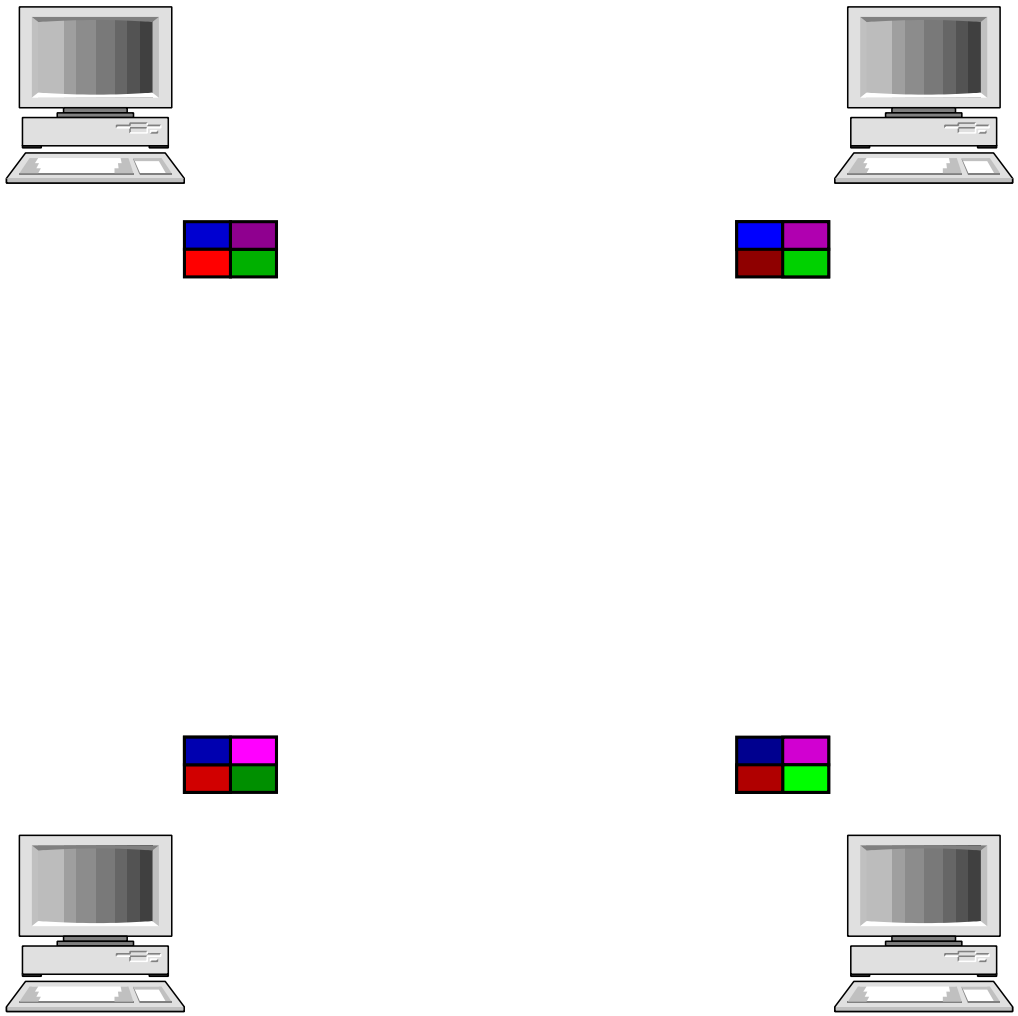
ATA Redukce: Změna struktury dat



ATA Redukce: Změna struktury dat



ATA Redukce: Změna struktury dat



ATA: Pro topologie Prsten či řetěz

Prsten

- 1. fáze: každý uzel pošle informaci svému sousedovi
- 2. až n-tá fáze: uzly sbírají a přeposílají příchozí zprávy
- Všechny linky jsou po celou dobu operace plně využité
- Optimální algoritmus

Řetěz

- Vysílání zpráv sousedům na obě strany
- Full-duplex linky $\Rightarrow n-1$ fází
- Half-duplex linky $\Rightarrow 2(n-1)$ fází

ATA Redukce

- Zprávy po jedné posílány po kruhu tak, že zpráva pro nejvzdálenější uzel je poslána jako první
- Při průchodu uzlem se přikombinuje lokální zpráva pro adresáta právě procházející zprávy

ATA: Pro topologie mřížky či hyperkostky

Mřížka

- 2 fáze – $\sqrt{p}$ řetízků o $\sqrt{p}$ uzlech
- Po první fázi uzly mají uzly $\sqrt{p}$ částí z celkového počtu p částí zprávy
- Příklad sítě 4x4, každý posílá své ID každému

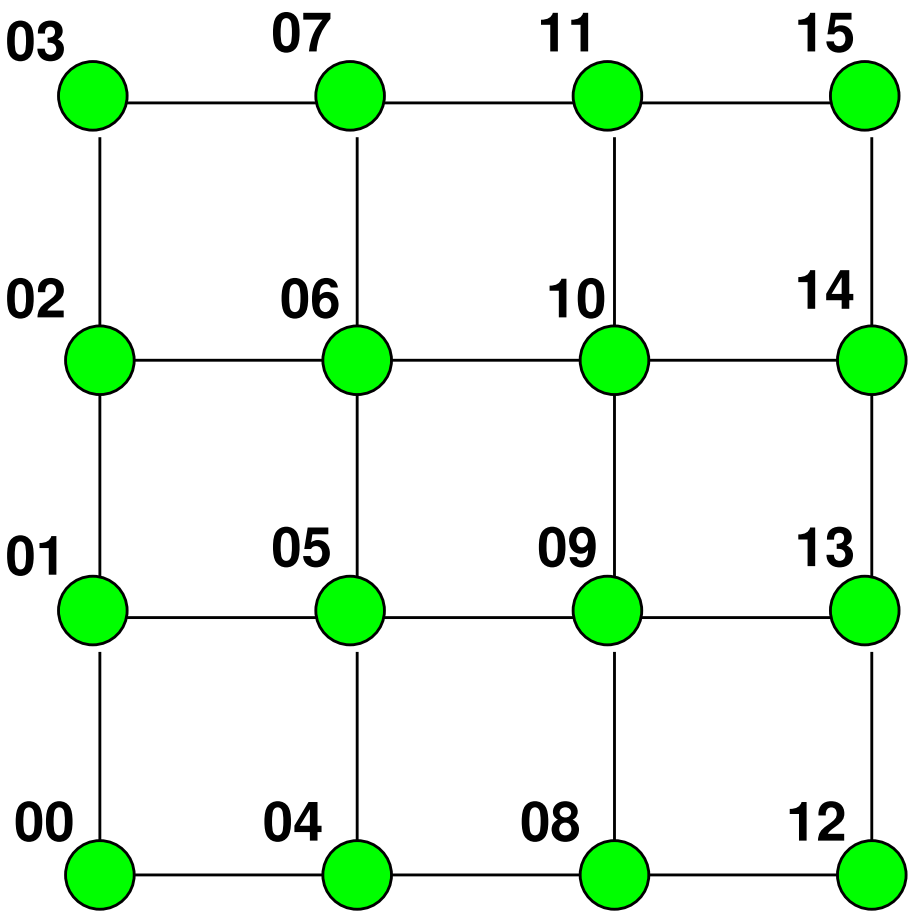
Hyperkostka

- Rozšíření algoritmu pro sítě na d dimenzí
- Po každé fázi (z celkového počtu d) je objem zpráv zdvojnásoben

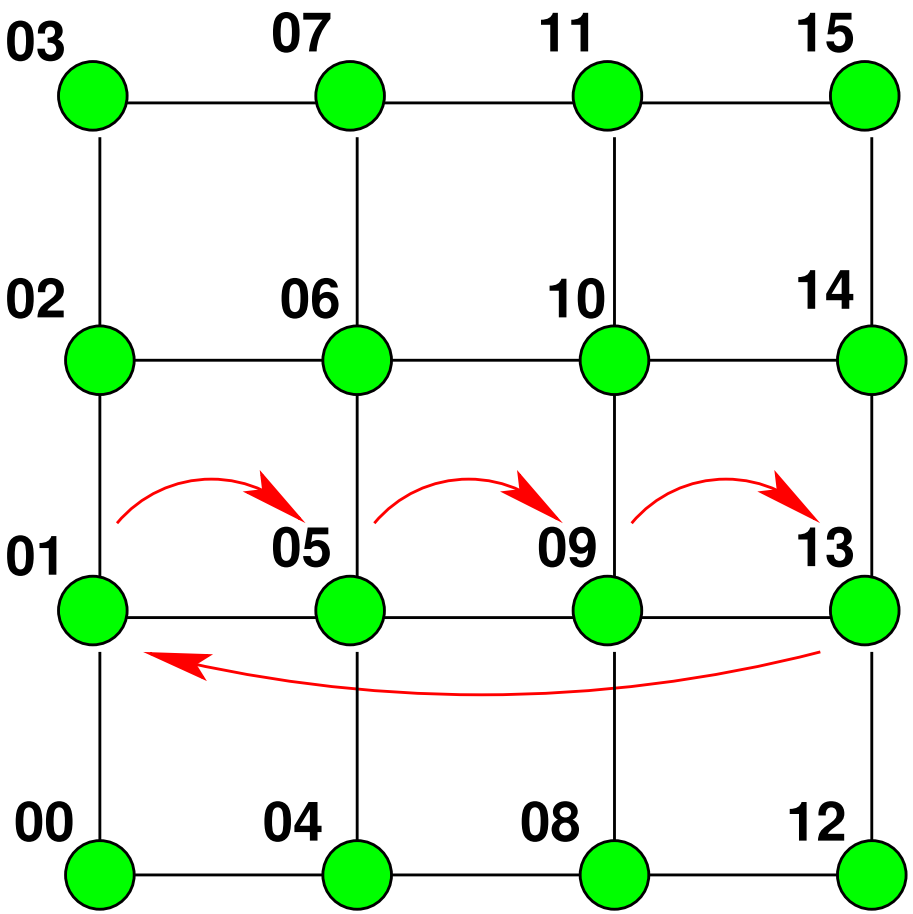
ATA Redukce

- Duální postup
- Po každé fázi je objem zpráv zredukován na polovinu

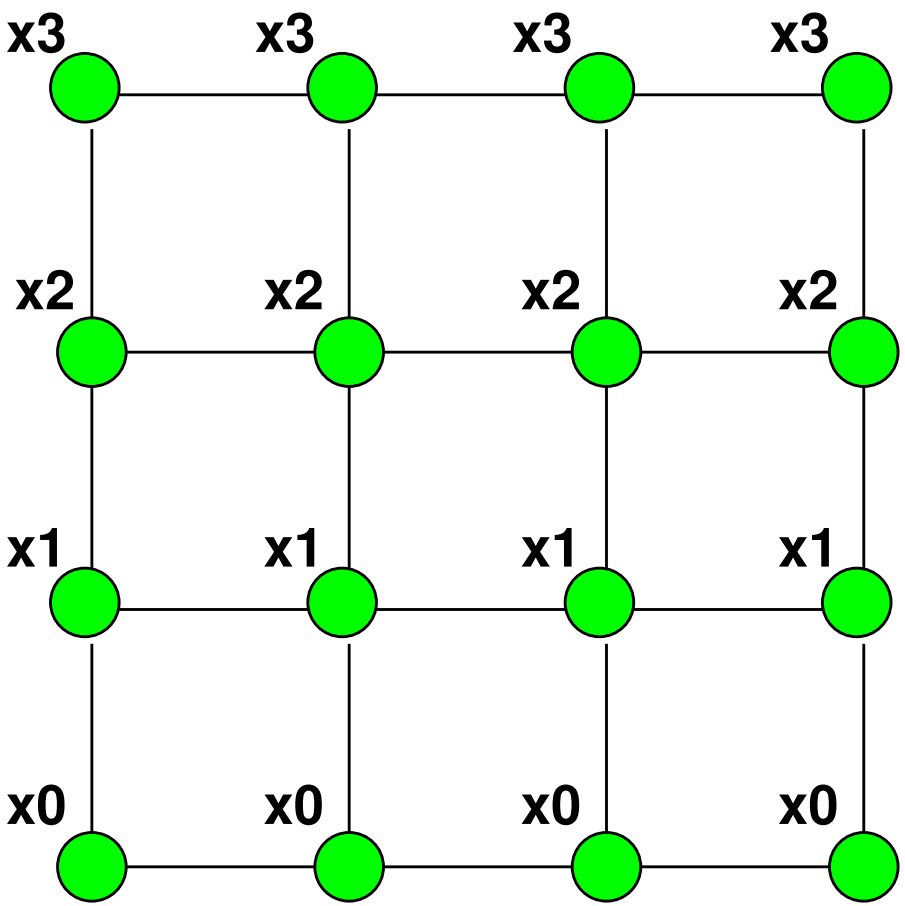
ATA Vysílání: Mřížka



ATA Vysílání: Mřížka



ATA Vysílání: Mřížka



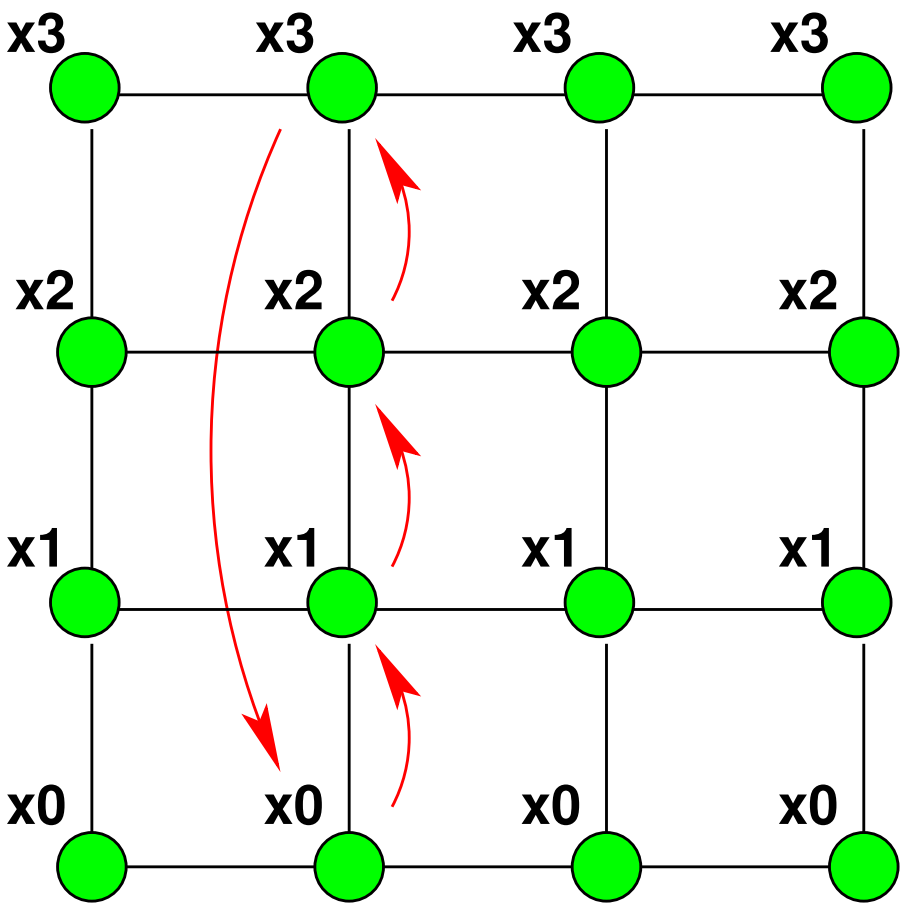
x3 = 03 07 11 15

x2 = 02 06 10 14

x1 = 01 05 09 13

x0 = 00 04 08 12

ATA Vysílání: Mřížka



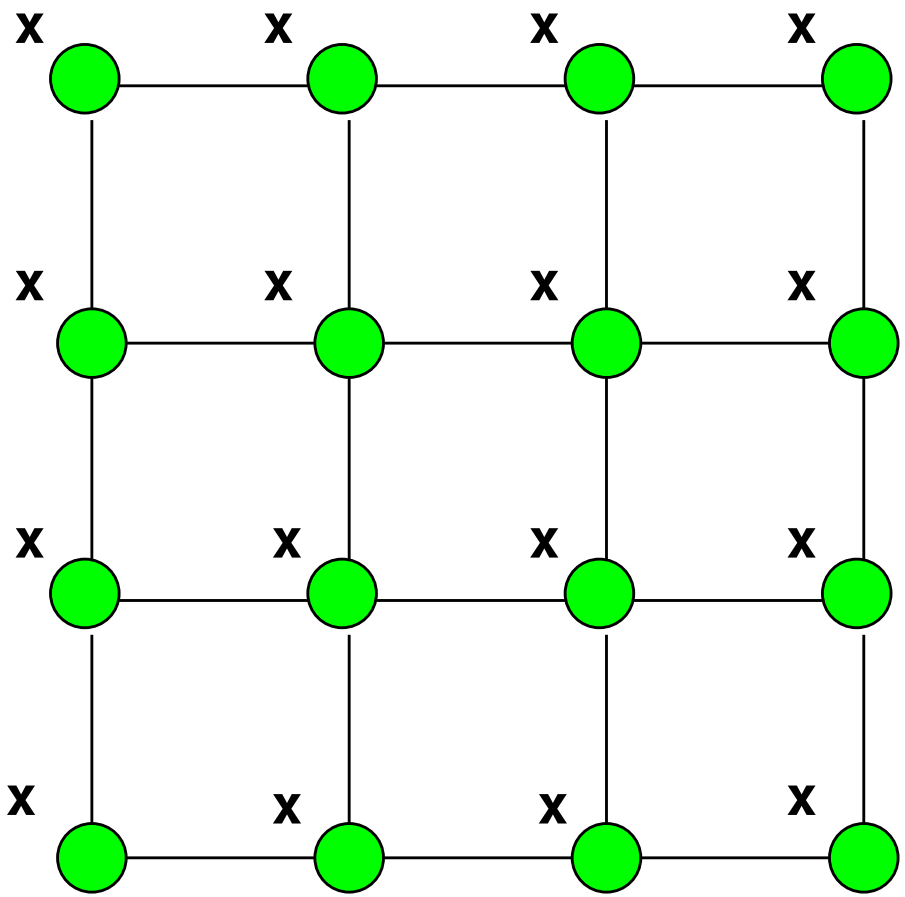
x3 = 03 07 11 15

x2 = 02 06 10 14

x1 = 01 05 09 13

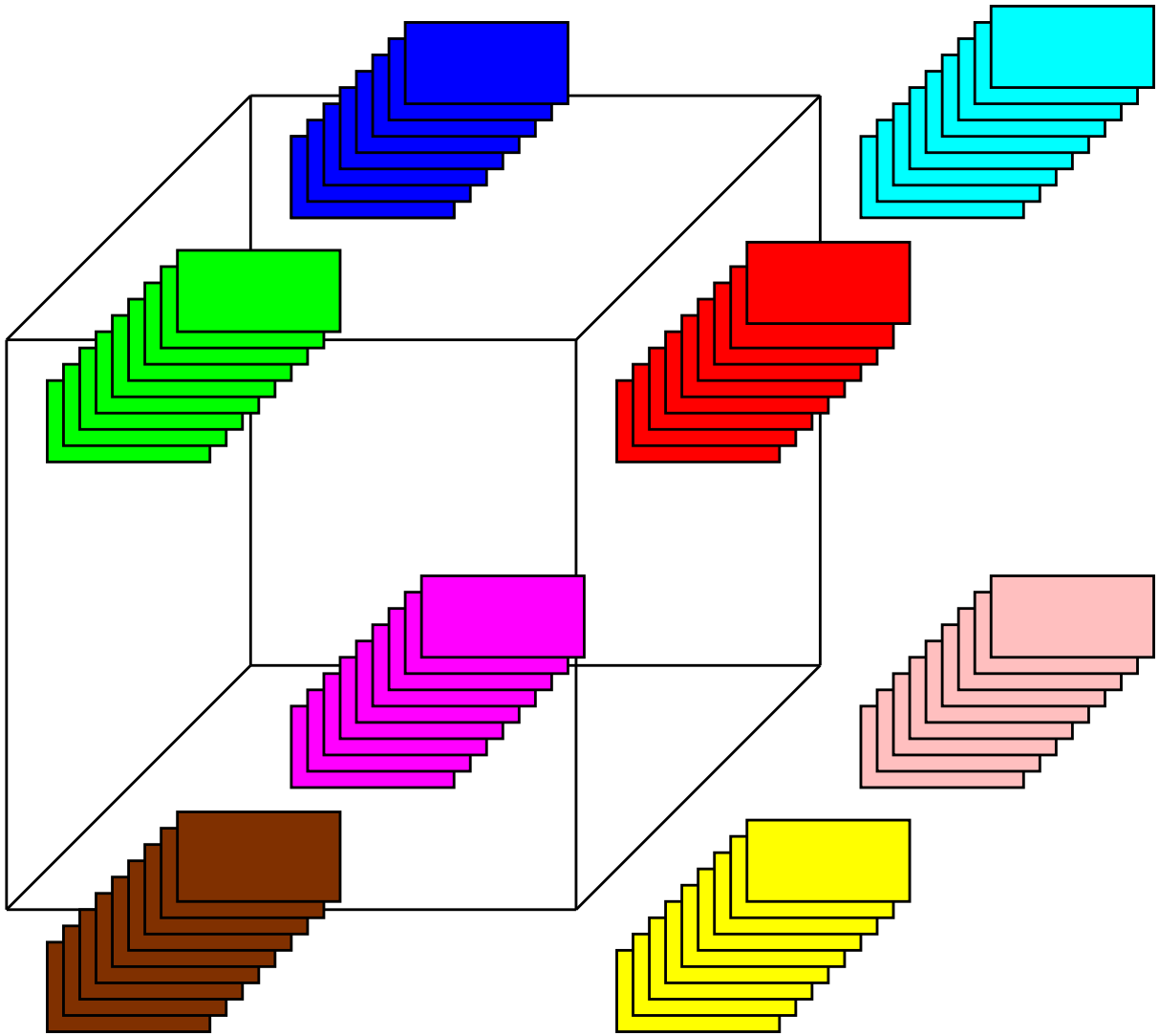
x0 = 00 04 08 12

ATA Vysílání: Mřížka

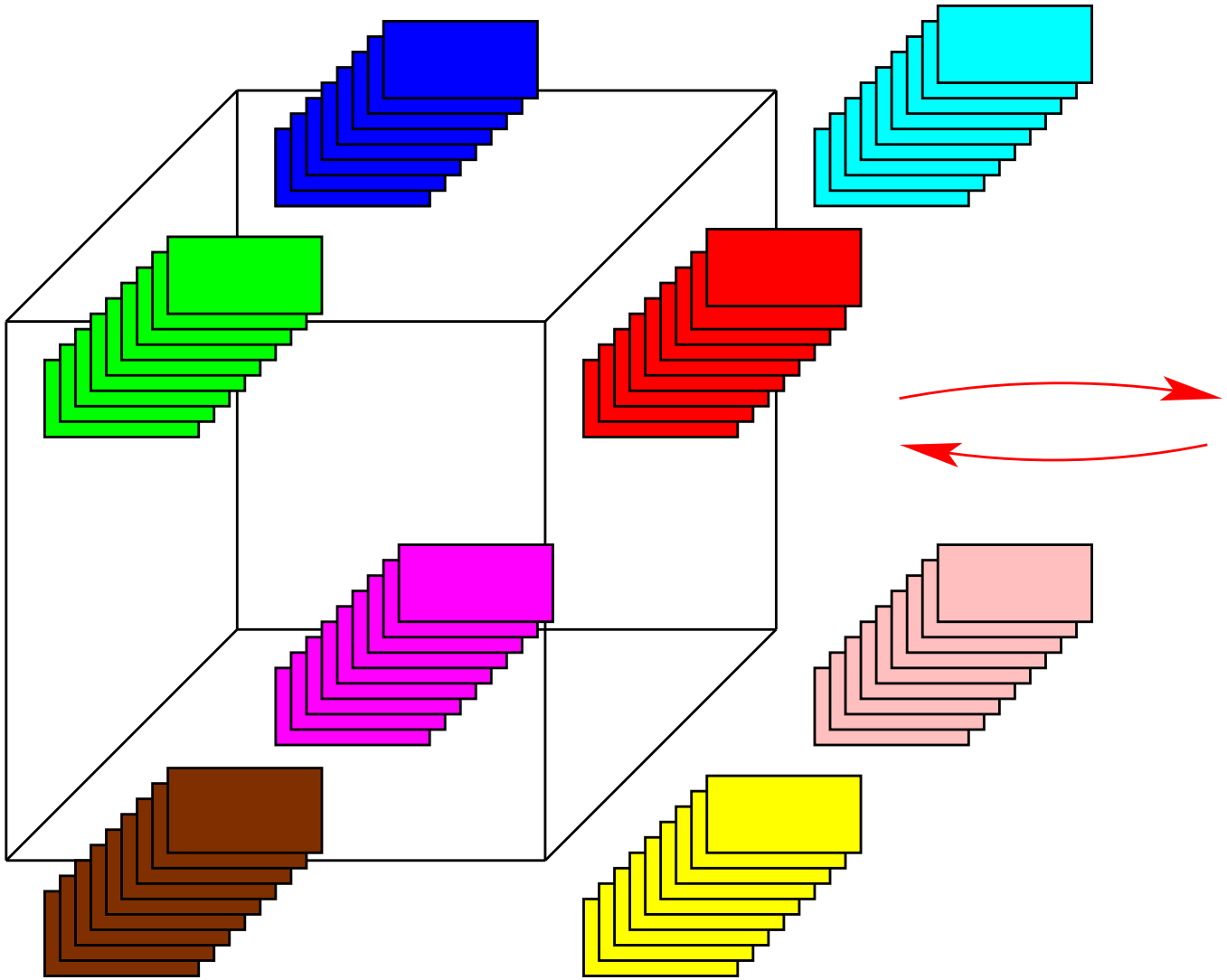


x = 03 07 11 15
02 06 10 14
01 05 09 13
00 04 08 12

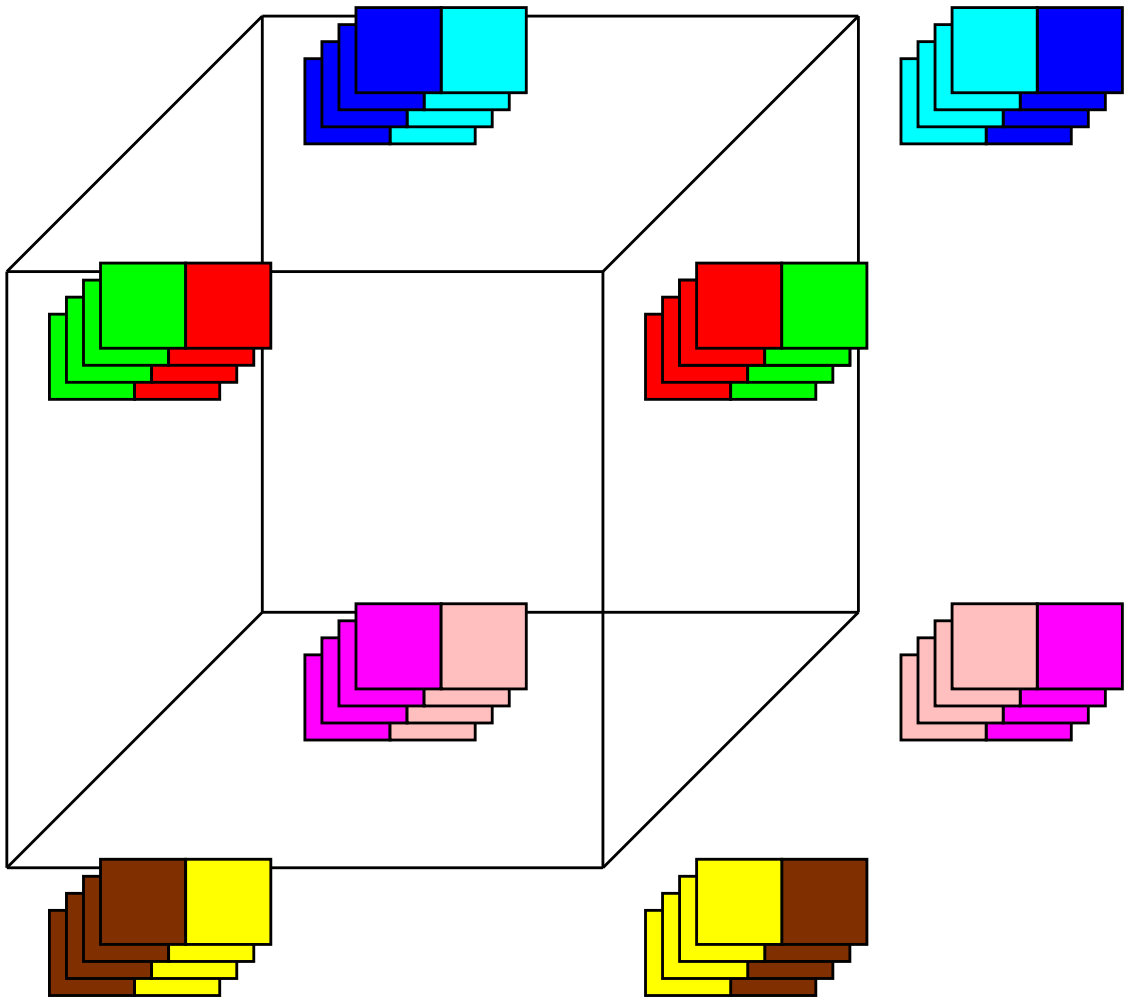
ATA Redukce: Hyperkostka



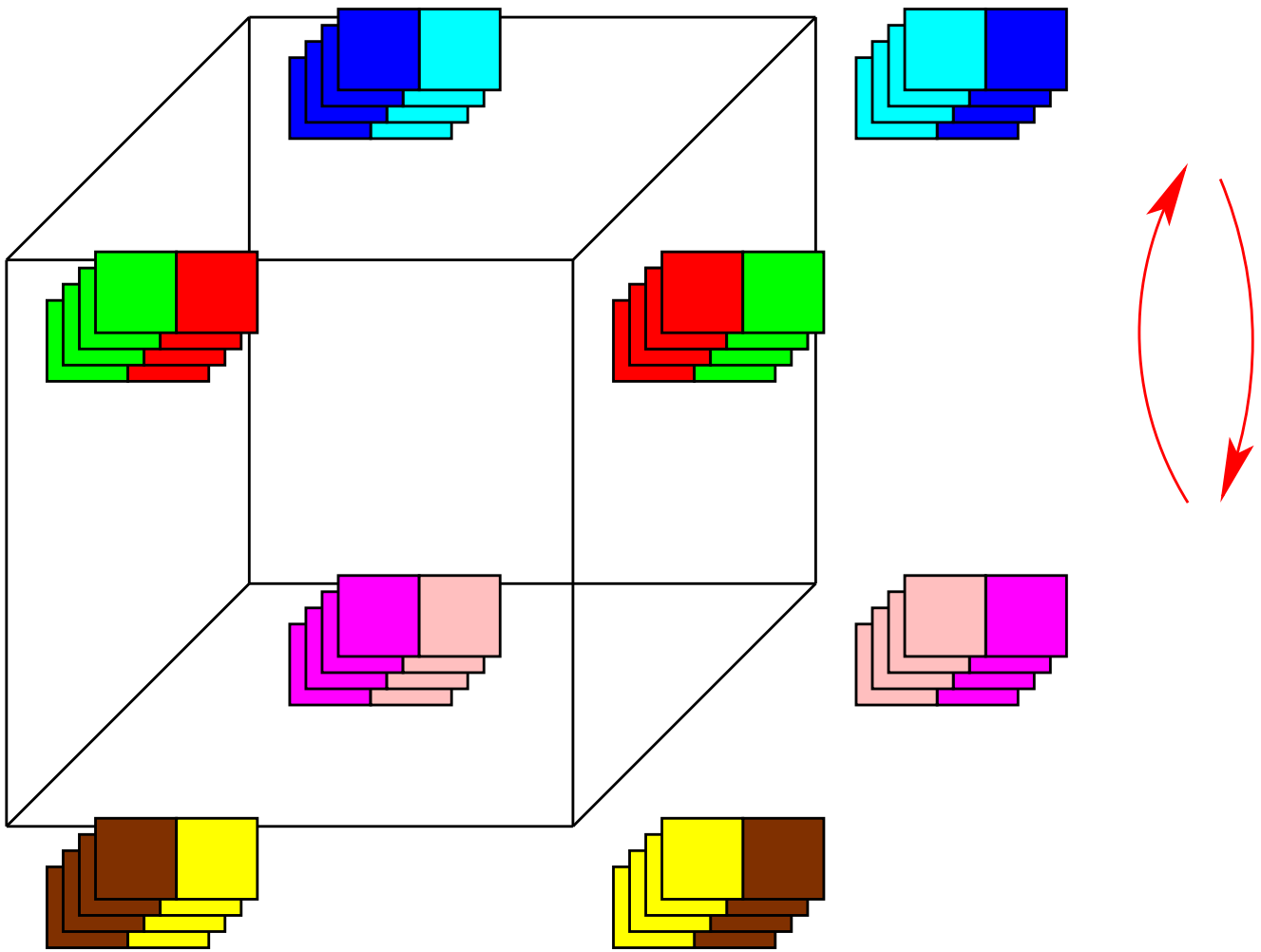
ATA Redukce: Hyperkostka



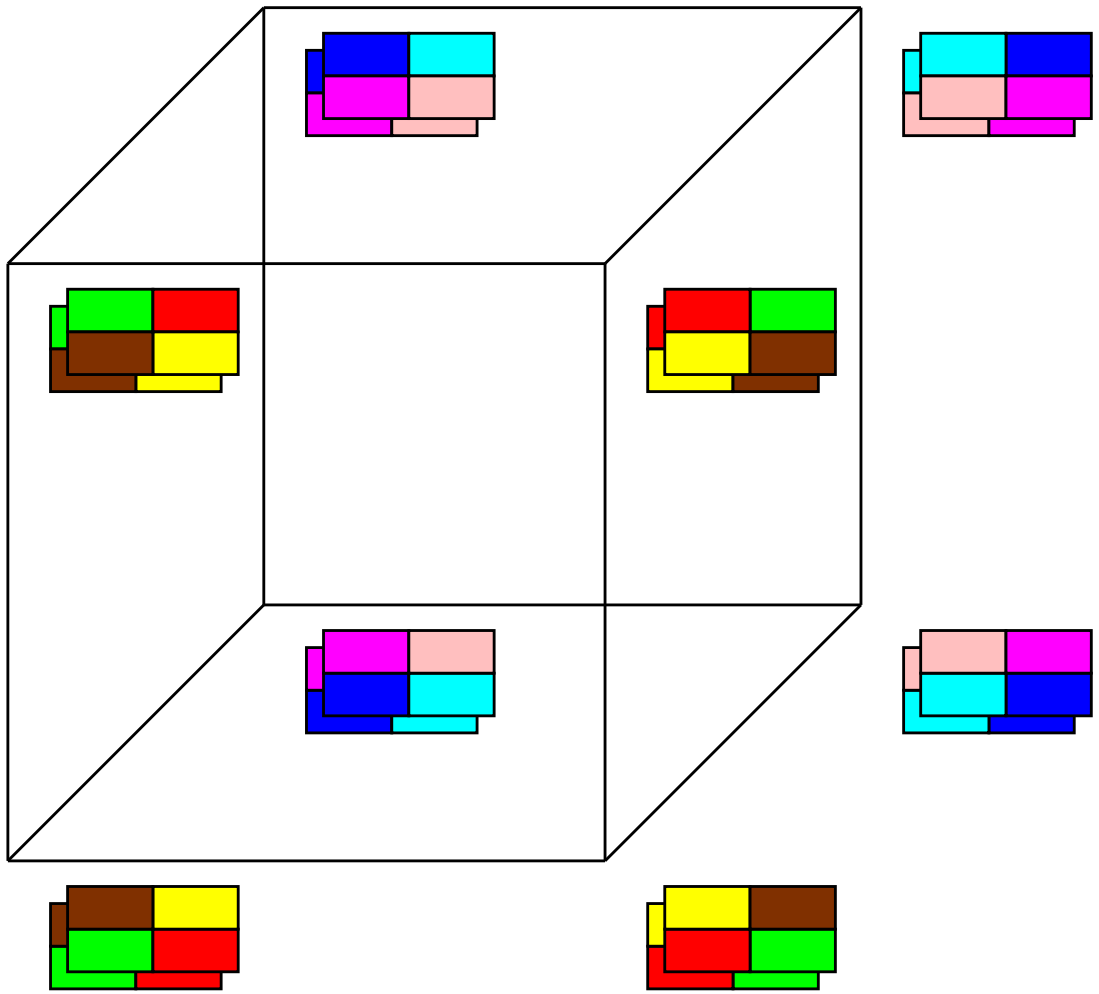
ATA Redukce: Hyperkostka



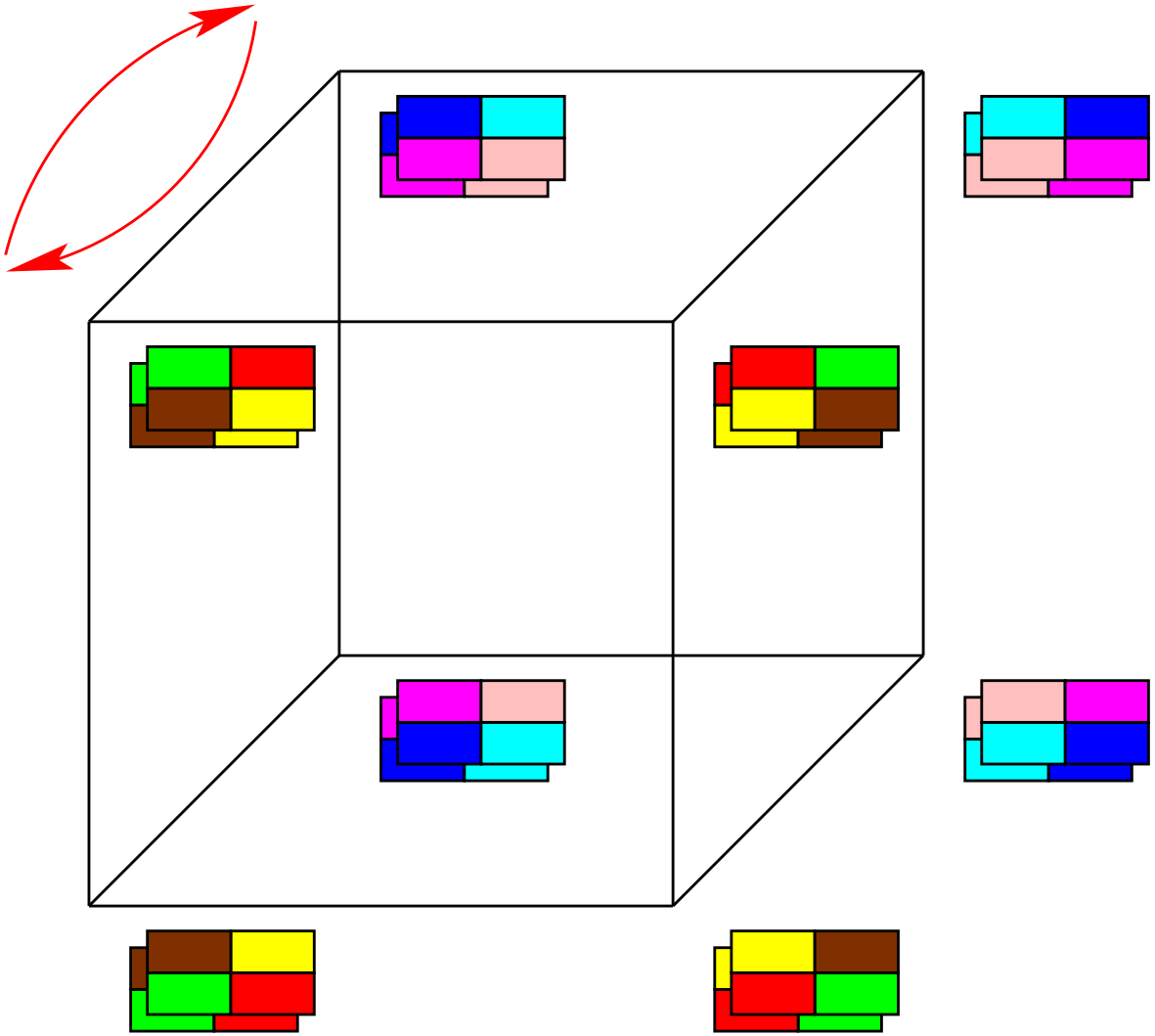
ATA Redukce: Hyperkostka



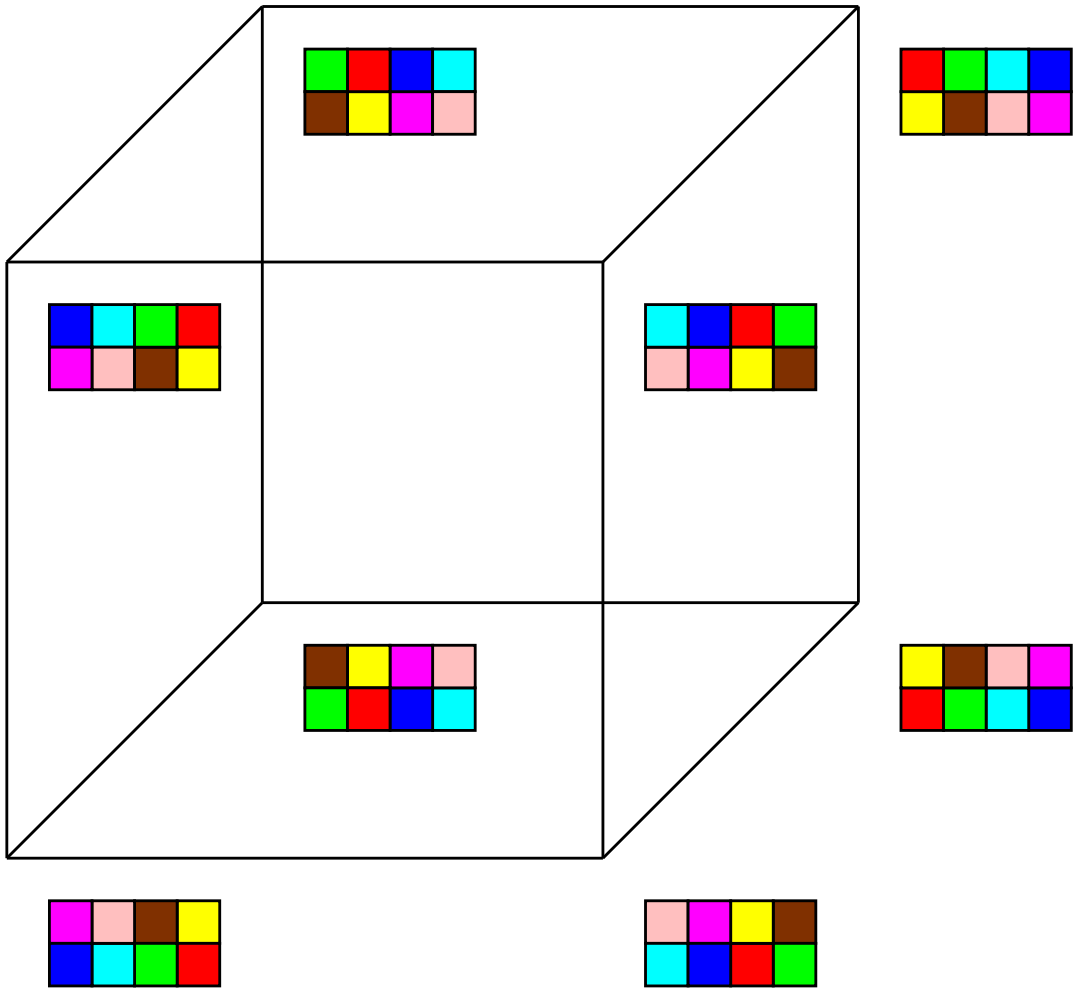
ATA Redukce: Hyperkostka



ATA Redukce: Hyperkostka



ATA Redukce: Hyperkostka



ATA: Analýza ceny

Kruh a lineární pole, p uzlů

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

2D mřížka, p uzlů

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

Hyperkostka, $p = 2^d$ uzlů

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

Pozorování

- Člen $t_w m(p-1)$ se vyskytuje vždy
- Pipeline několika OTA operací

Všichni všem individuální komunikace

Všichni všem individuální komunikace

All-To-All individuální komunikace (ATA individuální)

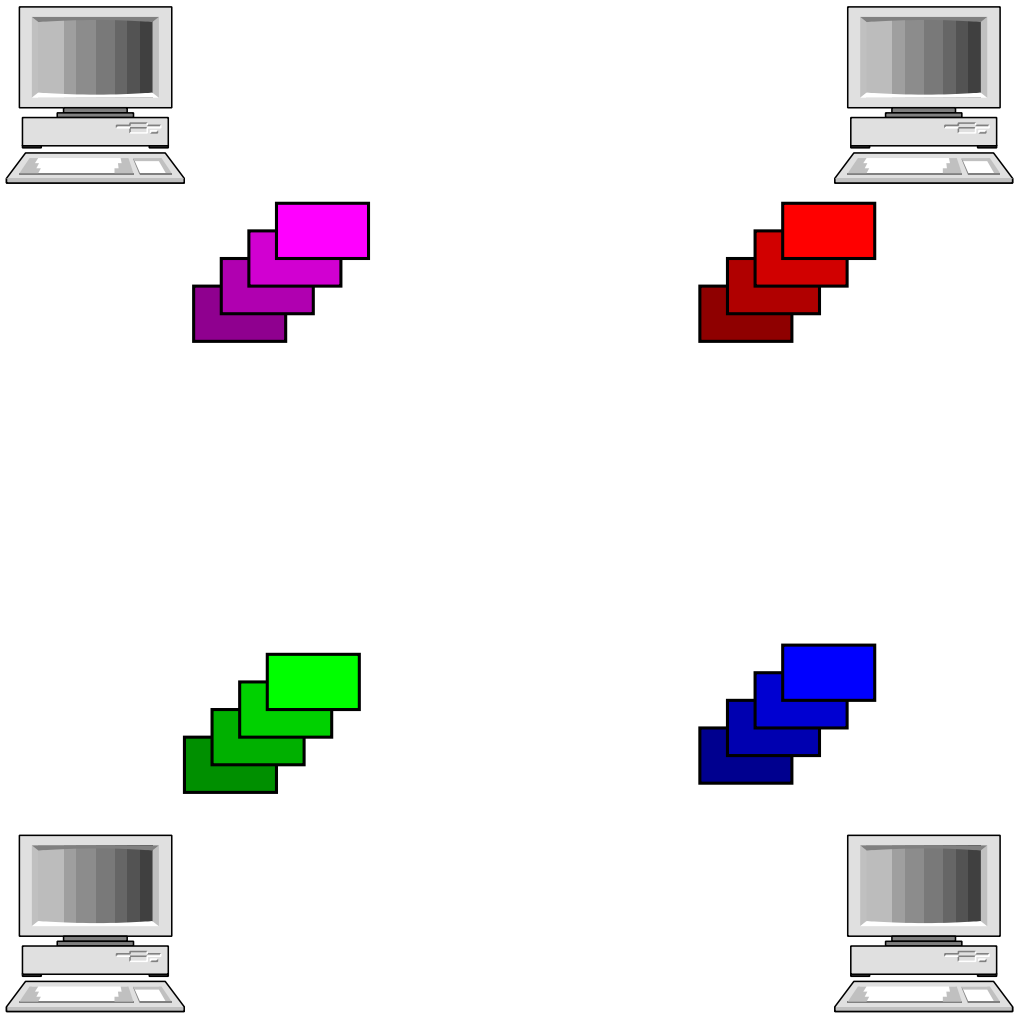
- Každá úloha posílá různá data ostatním úlohám
- Dojde k výměně 2D pole zpráv ($p \times p$) v 1D prostoru (p)
- Také označováno jako "totální výměna"
- Změna struktury dat:

$$p * (m_1, \ldots, m_p) \mapsto p * m_1, p * m_2, \ldots, p * m_p$$

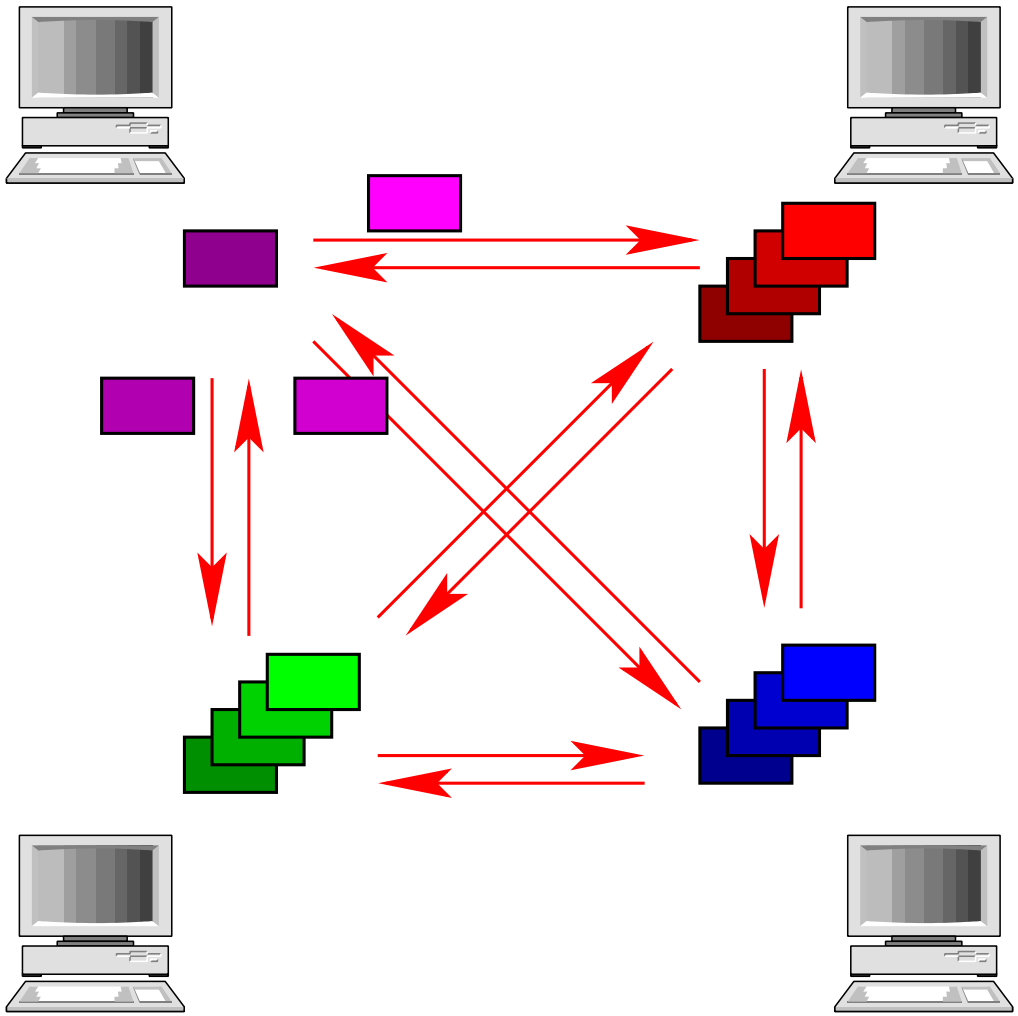
Příklad

- Transpozice matice ($A^T[i,j] = A[j,i]$)
- Matice $n \times n$ mapována po řádcích na n úloh
- Úloha j má k dispozici prvky $[j,0], [j,1], \ldots [j,n-1]$
- Úloha j chce znát prvky $[0,j], [1,j], \ldots [n-1,j]$

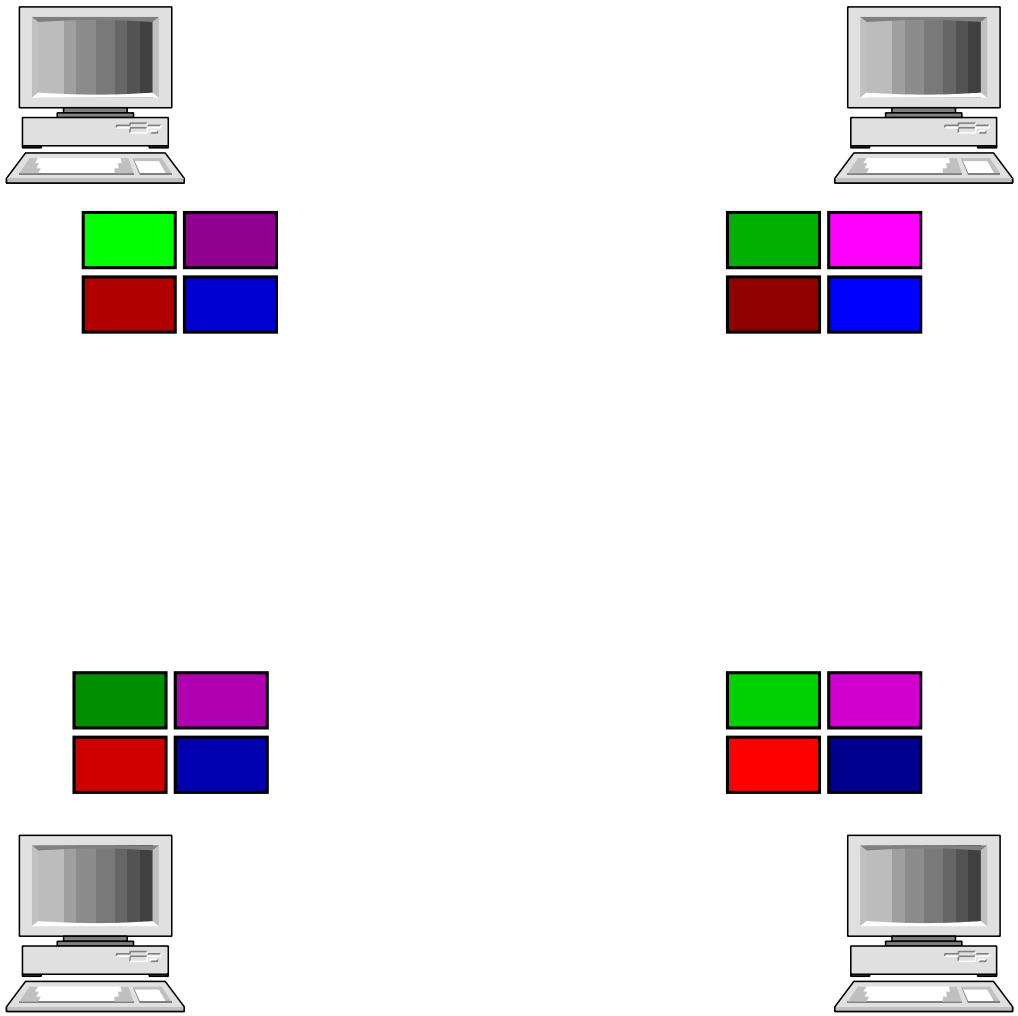
ATA Individuální: Změna struktury dat



ATA Individuální: Změna struktury dat



ATA Individuální: Změna struktury dat



ATA Individuální: Pro topologie Prsten či Řetěz

Princip

- Nejprve všechny úlohy pošlou jedním směrem zprávy pro zbývajících $p-1$ úloh
- V každém dalším kole každá úloha vyextrahuje zprávu, která je určena jí a zbývající zprávy přepošle
- V posledním kole všechny úlohy přeposílají pouze jednu zprávu

Analýza ceny

- Počet kol je $p-1$, všechny zprávy mají velikost m
- $T = \sum_{i=1}^{p-1}(t_s + t_w m(p-i))$
  $= t_s(p-1) + \sum_{i=1}^{p-1} i t_w m$
  $= (t_s + t_w mp/2)(p-1)$

ATA Individuální: Pro topologii mřížky

Princip

- Síť je $\sqrt{p}$ řetízků o délce $\sqrt{p}$
- 1. fáze: mezi řetízky se vymění v jednom směru zprávy tak, aby informace pro každý uzel v řetízku byla někde v řetízku obsažena
- 2. fáze: v rámci řetízku se informace napropaguje na odpovídající místo

Příklad

- ATA Individuální komunikace na síti 4x4 uzly

Cena

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

ATA Individuální: Pro topologii hyperkostky

Princip

- Aplikace algoritmu pro d-dimenzionální sítě
- Podél jedné dimenze se posílá vždy $p/2$ zpráv

Příklad

- Krychle $2 \times 2 \times 2$ uzlů

Cena

- V každé fázi vyměněno $mp/2$ dat
- $log\ p$ fází

$$T = (t_s + t_w mp/2) log\ p$$

- Navíc v každé fázi uzly přeuspořádávají/třídí zprávy

ATA Individuální: Pro topologii hyperkostky – optimalita

Problém

- Každý uzel posílá a přijímá $m(p-1)$ dat
- Průměrná vzdálenost, na kterou data putují je $(log\ p)/2$
- Celkový provoz na síti je $p \times m(p-1) \times (log\ p)/2$
- Počet linek v hyperkostce je $p(log\ p)/2$
- Optimální algoritmus by měl dosáhnout složitosti

$$T = \frac{t_w pm(p-1)(log\ p)/2}{(plog\ p)/2} = t_w m(p-1)$$

Závěr

- Pro ATA Individuální komunikaci není algoritmus pro sítě ve tvaru mříží optimální na sítích ve tvaru hyperkostky

ATA Individuální: Optimální algoritmus pro hyperkostku

```
(1) proc All-To-All-Personal(d, id)
(2)     for i := 1to 2^d − 1
(3)         partner := id XOR i
(4)         send M_id to partner
(5)         receive M_partner from partner
(6)     end
(7) end
```

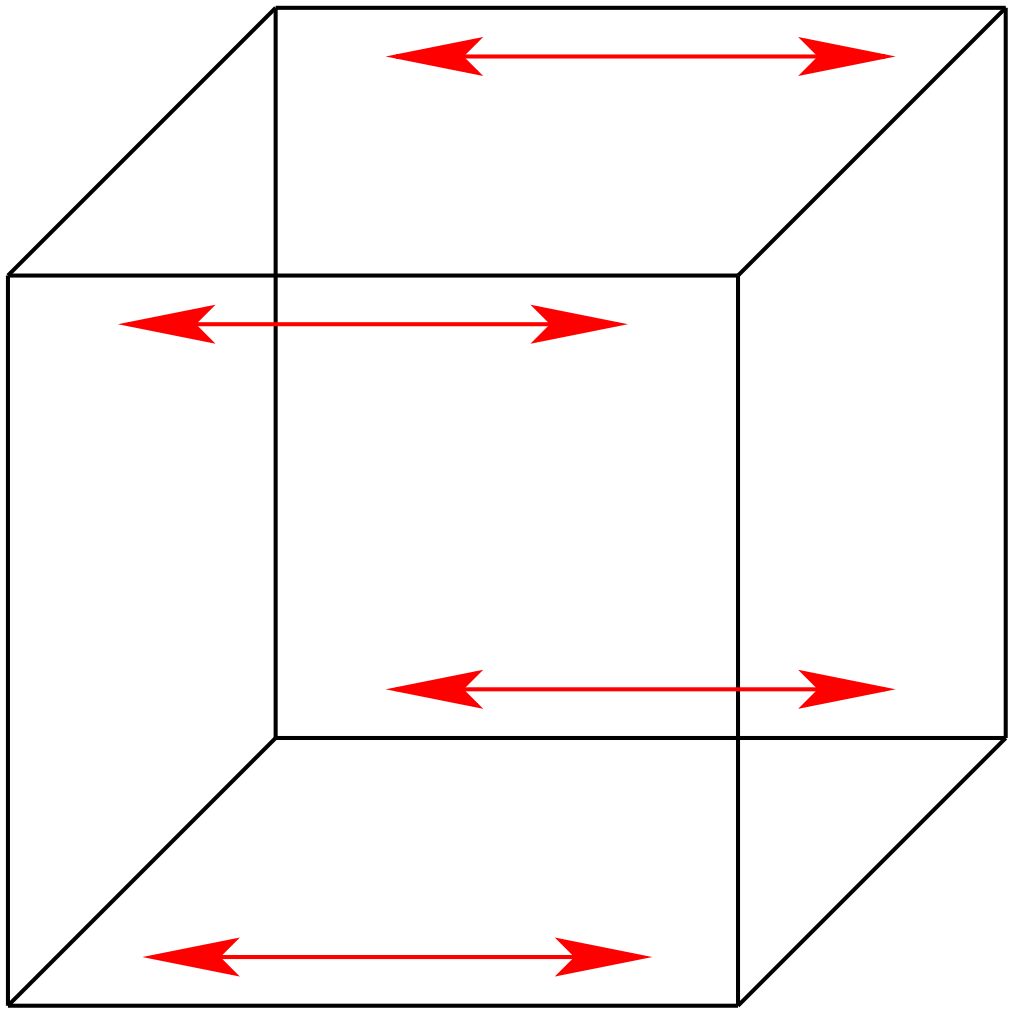
Pozorování:

- Systematická a symetrická volba partnera
- Lze routovat tak, aby nedocházelo k blokování linek
- E-cube routing: Cesta mezi 2 body je dána pozicí odlišných bitů těchto bodů, routuje se od nejméně významného bitu
- Vyžaduje duplexní linky

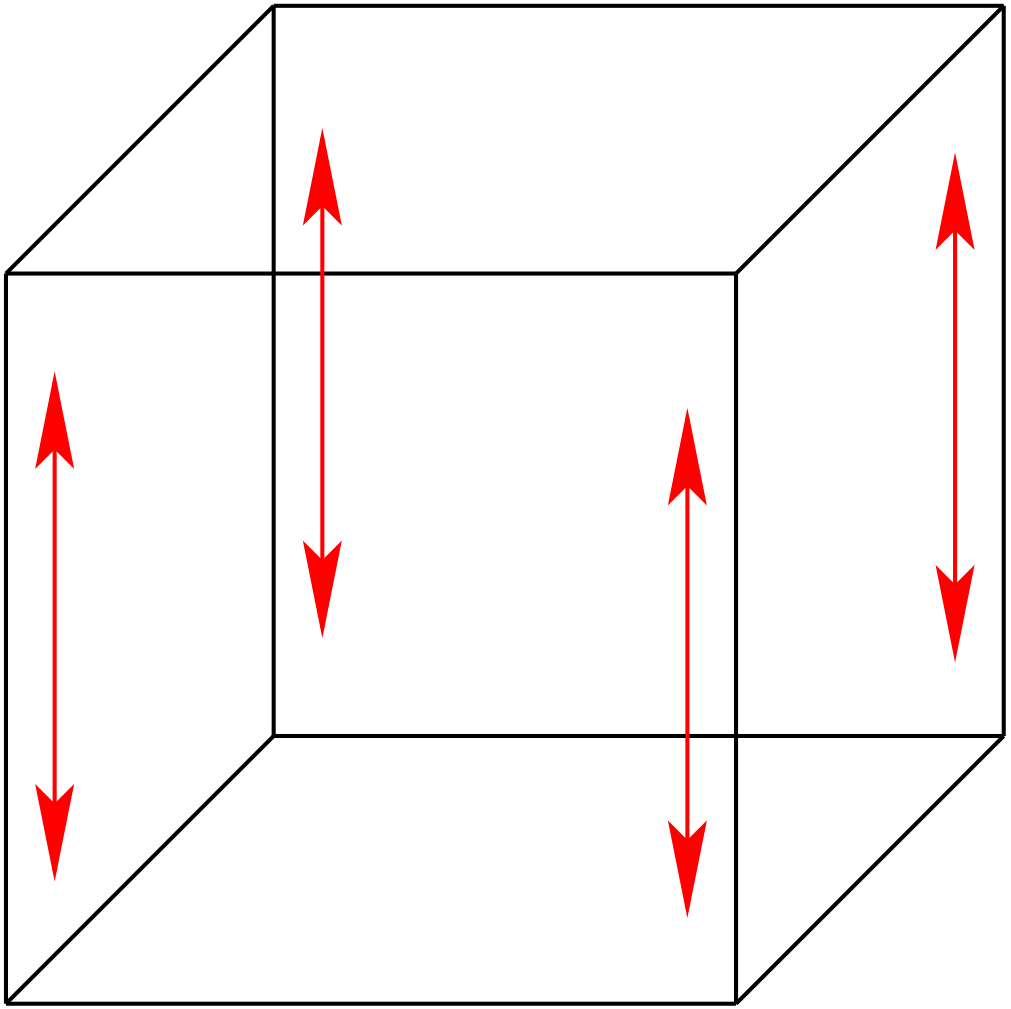
Cena:

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

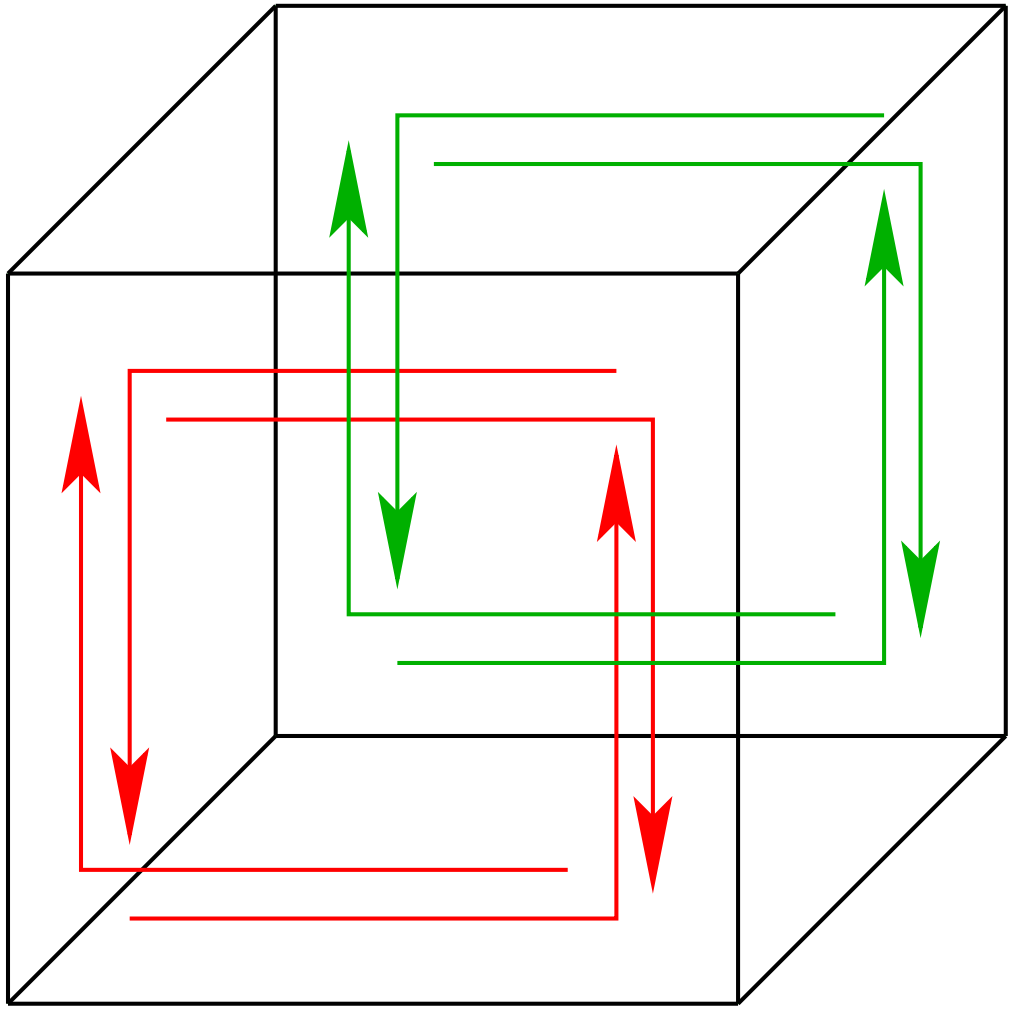
E-Cube routing



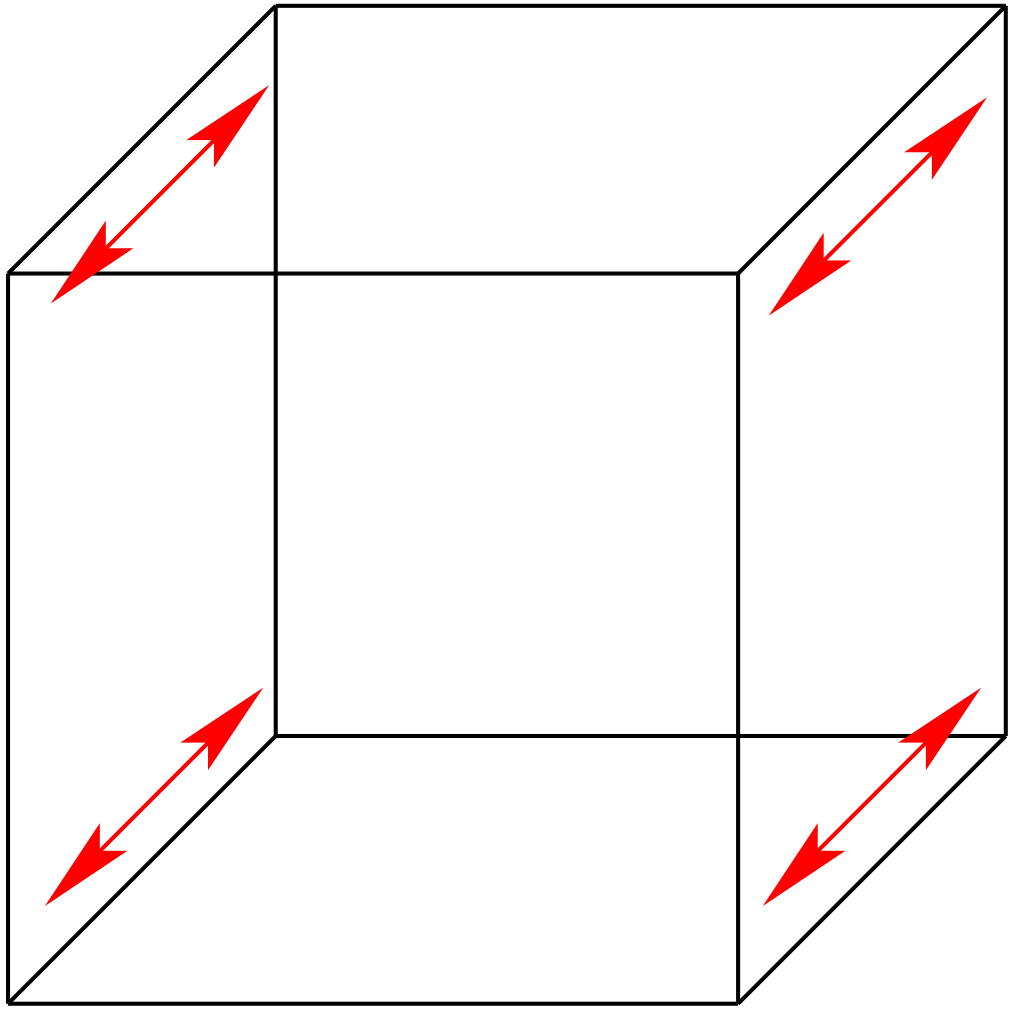
E-Cube routing



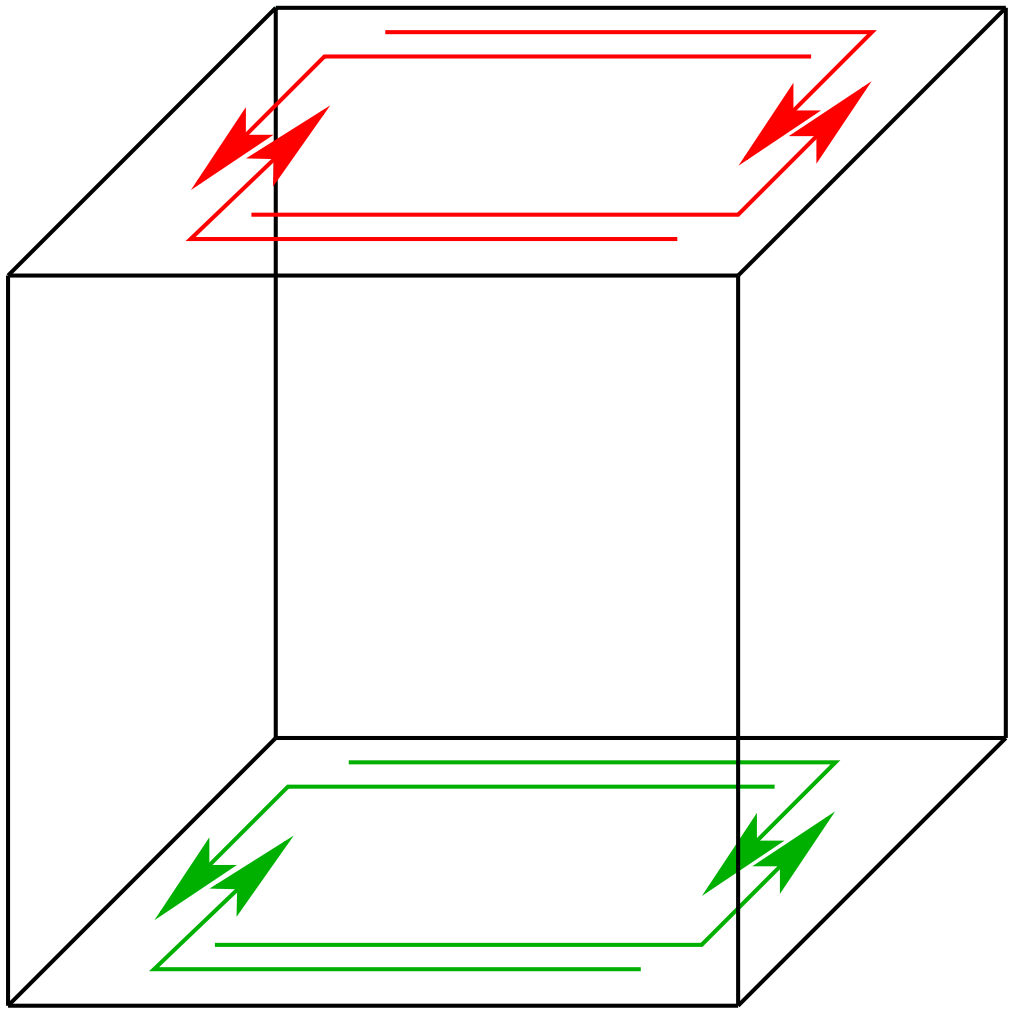
E-Cube routing



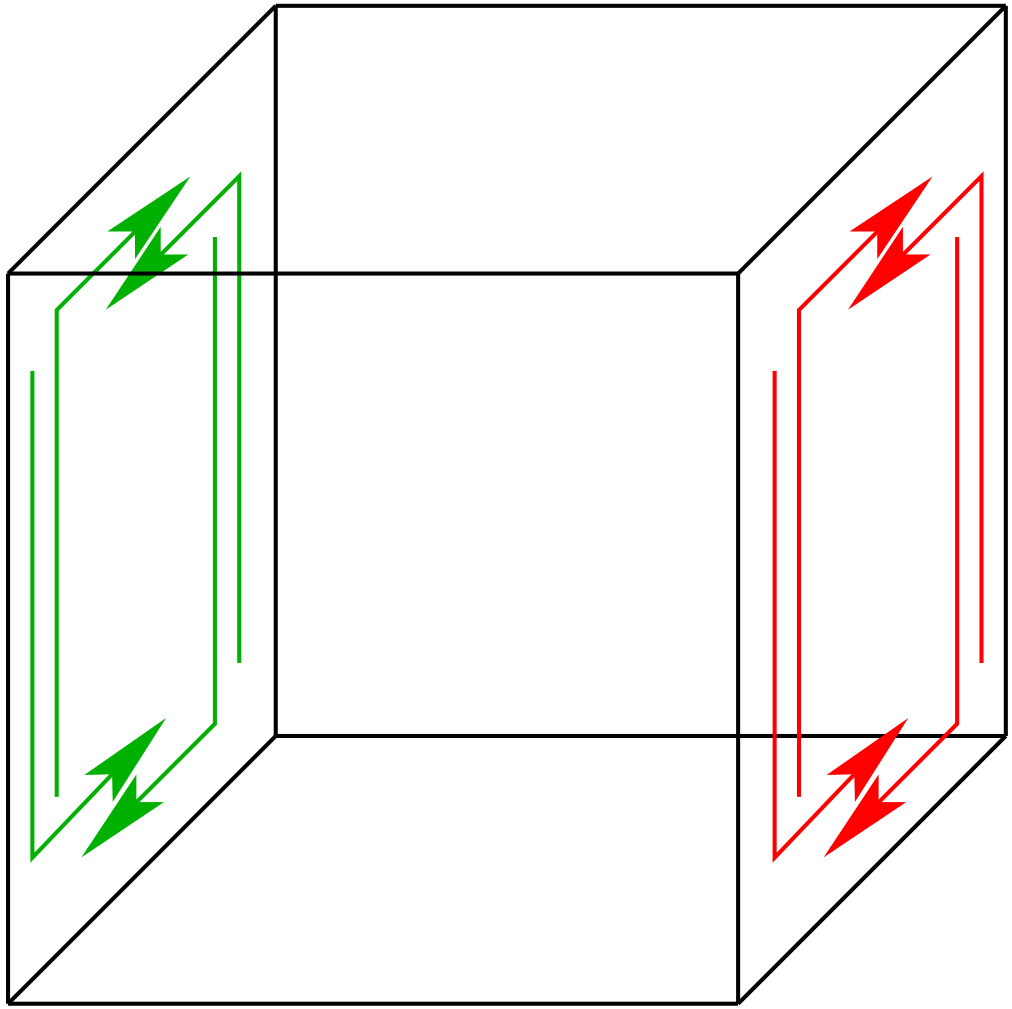
E-Cube routing



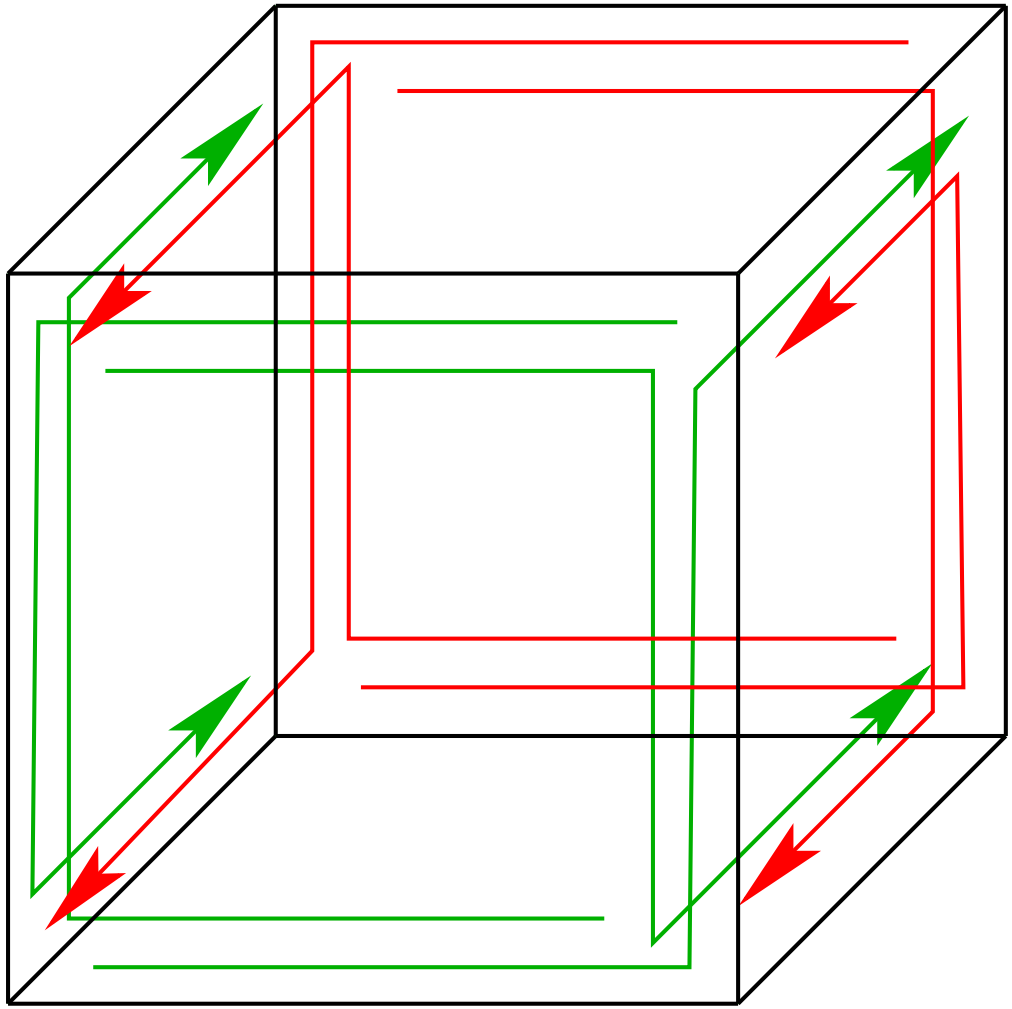
E-Cube routing



E-Cube routing



E-Cube routing



Operace kompletní redukce a prefixového součtu

Operace kompletní redukce

Kompletní redukce (All reduce)

- Úlohy si vzájemně vyměňují data, která se kombinují
- Lze realizovat jako ATO redukci následovanou OTA vysíláním výsledku z předchozí redukce
- Změna struktury dat: $p * m \mapsto p * m$

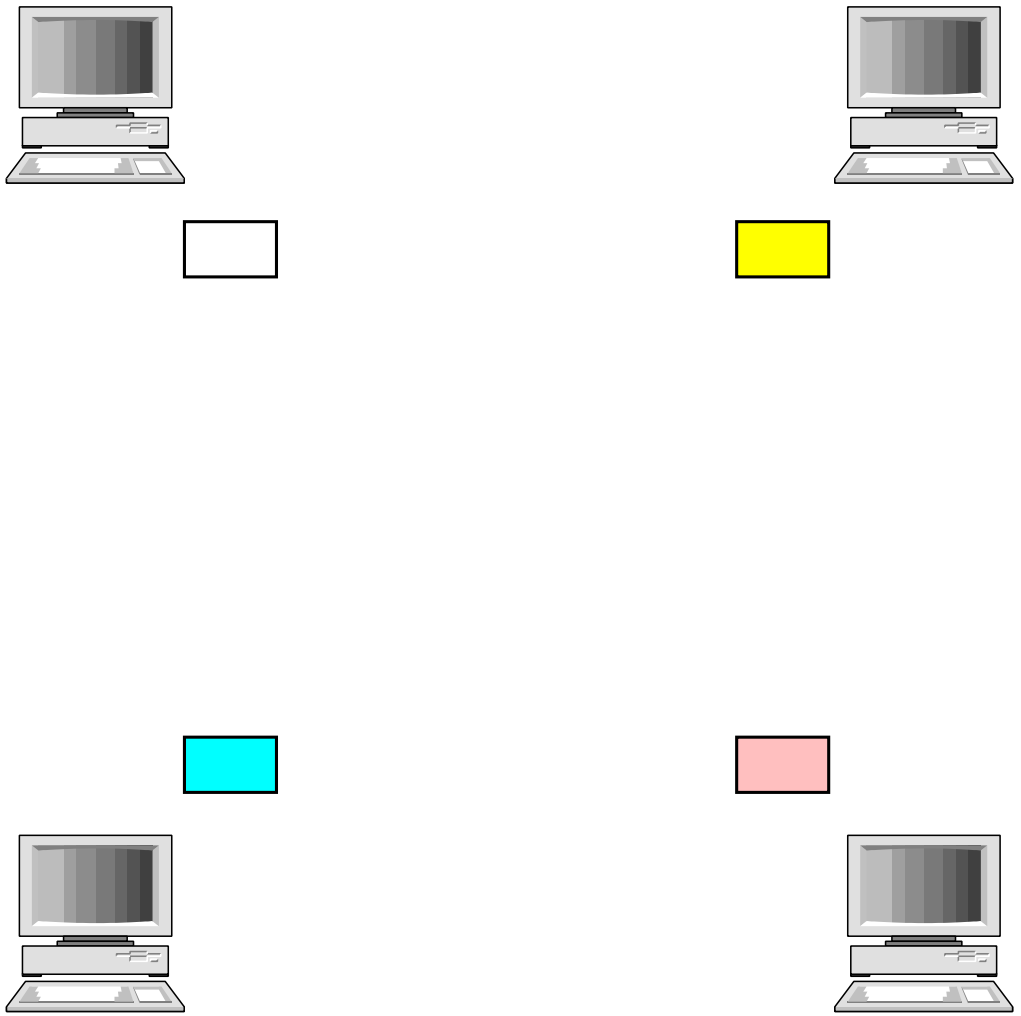
Sémantika a využití pro účely synchronizace

- Úloha nemůže dokončit operaci, dokud všechny participující úlohy nepřispějí svým dílem
- Může být použito pro realizaci synchronizačního primitiva

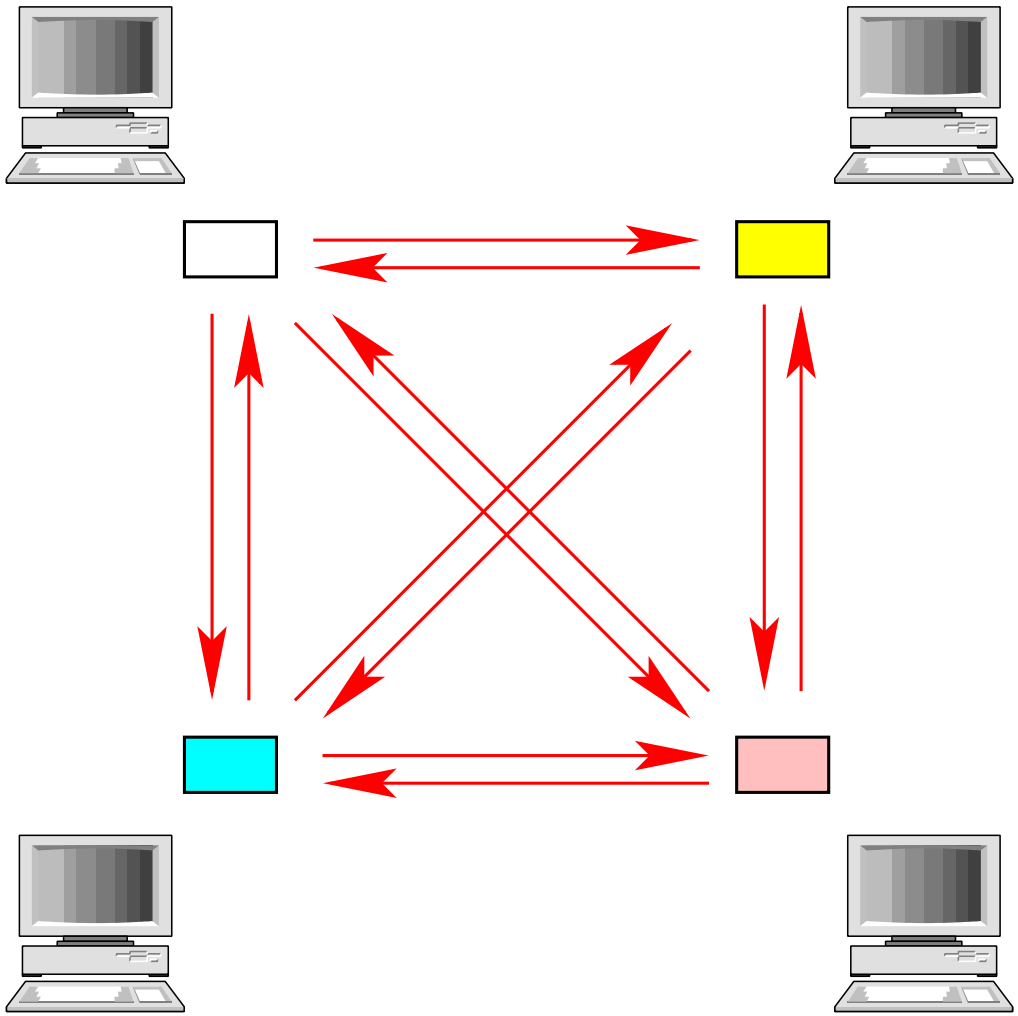
Implementace

- Naivně pomocí ATO a OTA, nebo
- Modifikací operace ATA vysílání
  - rozdíl od ATA: zprávy se nekumulují, ale kombinují
- Cena pro hyperkostku je $T = (t_s + t_w m) log\, p$

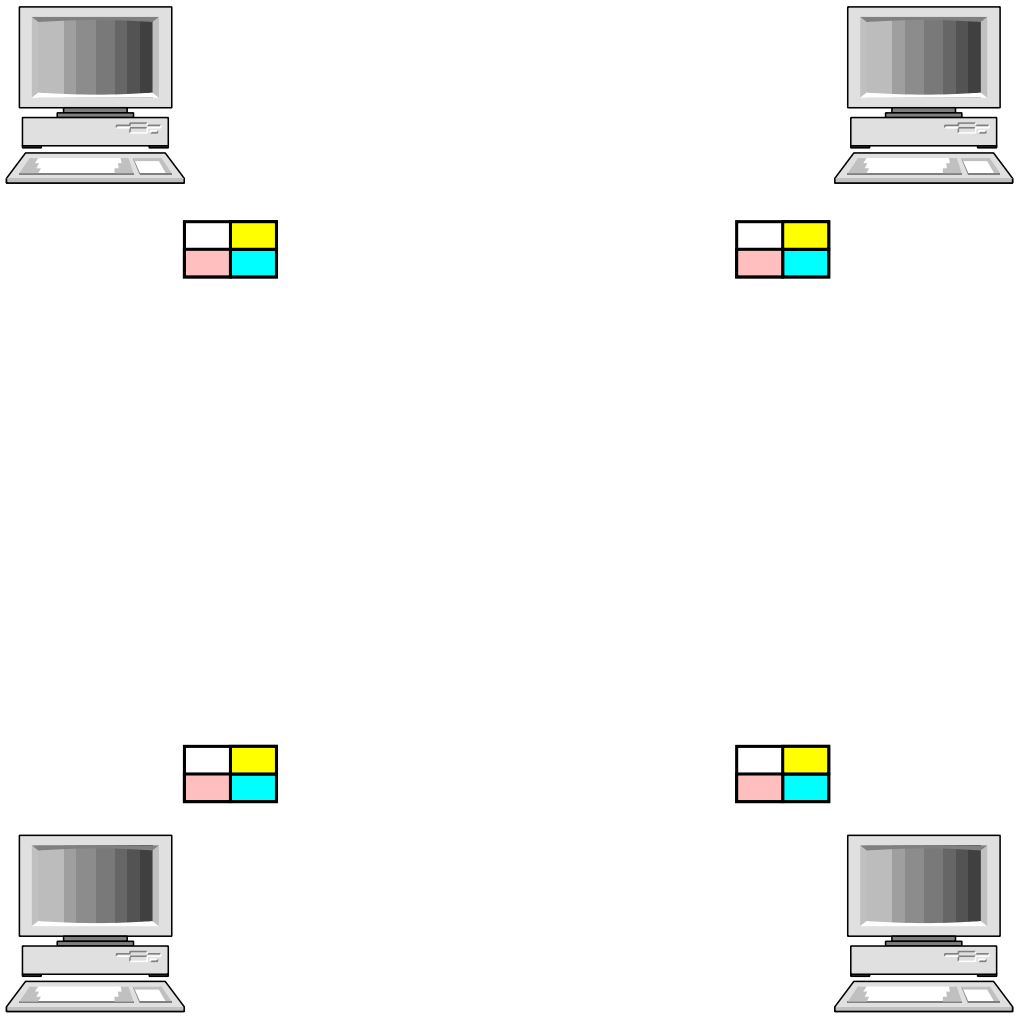
Kompletní redukce: Změna struktury dat



Kompletní redukce: Změna struktury dat



Kompletní redukce: Změna struktury dat



Operace prefixového součtu

Prefixový součet

- Úlohy posílají data ostatním úlohám se stejným či menším ID
- Data se kombinují (redukce)
- Změna struktury dat: $p * m \mapsto p * m$

Příklad

- Data v jednotlivých uzlech: $\langle 3, 1, 4, 0, 2 \rangle$
- Kombinace pomocí operace součtu
- Výsledná data $\langle 3, 4, 8, 8, 10 \rangle$

Implementace

- Použití algoritmu pro ATA
- Jak se pozná, že zpráva pochází od uzlu s menším ID?
- Všechny posílaná data obohacena o identifikátor původce

Scatter and Gather

Scatter and Gather

Scatter (též označováno jako "Scan")

- Jedna úloha posílá unikátní zprávu každé další úloze
- One-To-All individuální (personalized) komunikace
- Na rozdíl od OTA vysílání se žádná data neduplikují
- Změna struktury dat: $(m_1, \ldots, m_p) \mapsto m_1, \ldots, m_p$

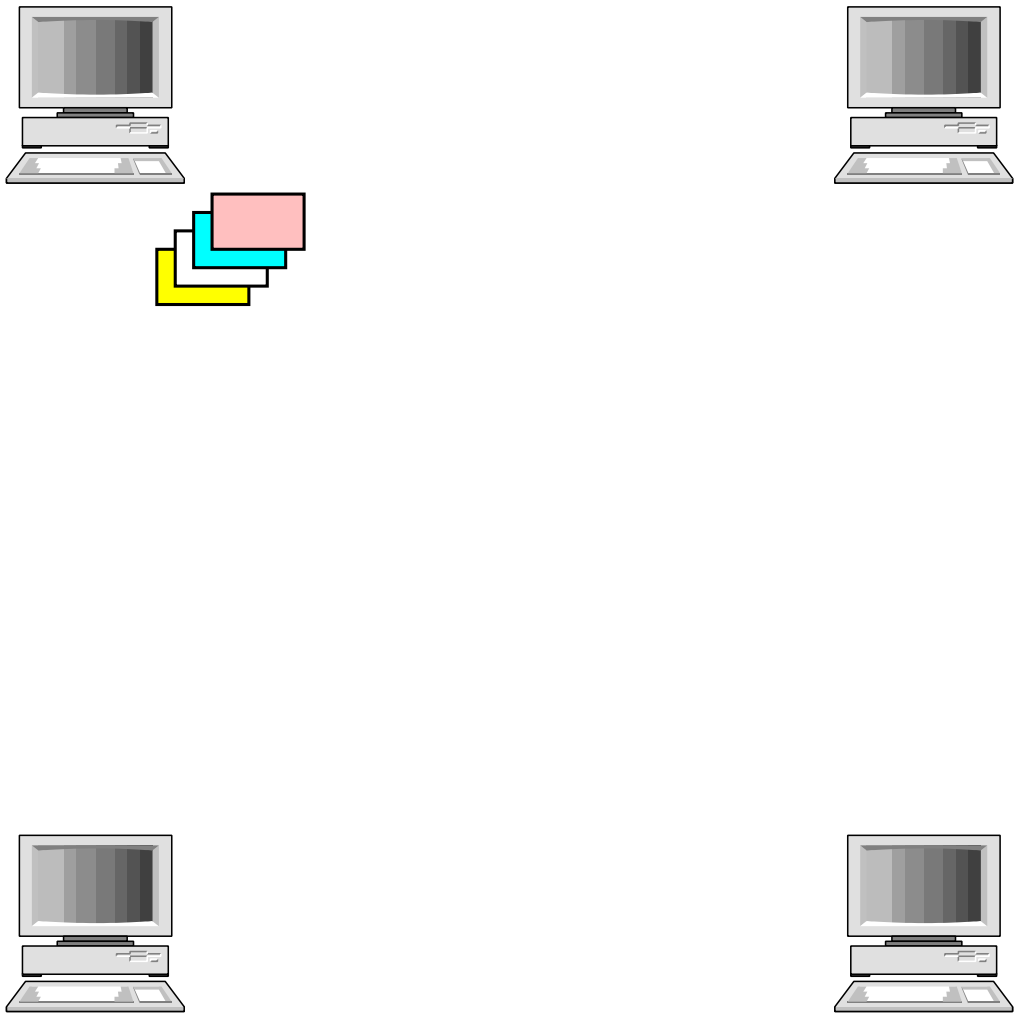
Gather (též označováno jako "Concatenation")

- Jedna úloha sbírá unikátní data od ostatních úloh
- All-To-One individuální (personalized) komunikace
- Na rozdíl od ATO redukce se nevyskytuje kombinace dat
- Změna struktury dat: $m_1, \ldots, m_p \mapsto (m_1, \ldots, m_p)$

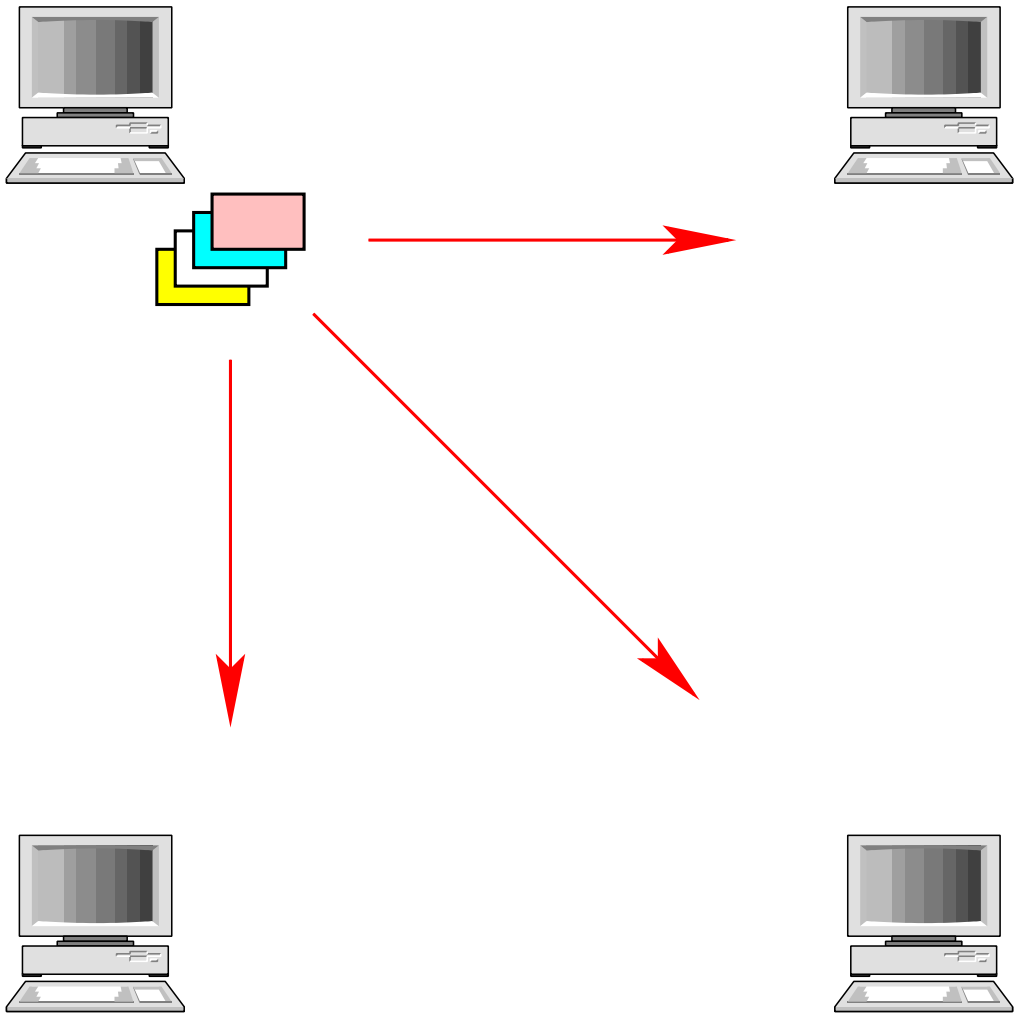
Implementace

- Využití algoritmů pro ATO a OTA
- Celkem $log\ p$ fází, s každou fází se objem dat zvětšuje
- $T = t_s(log\ p) + t_w m(p-1)$

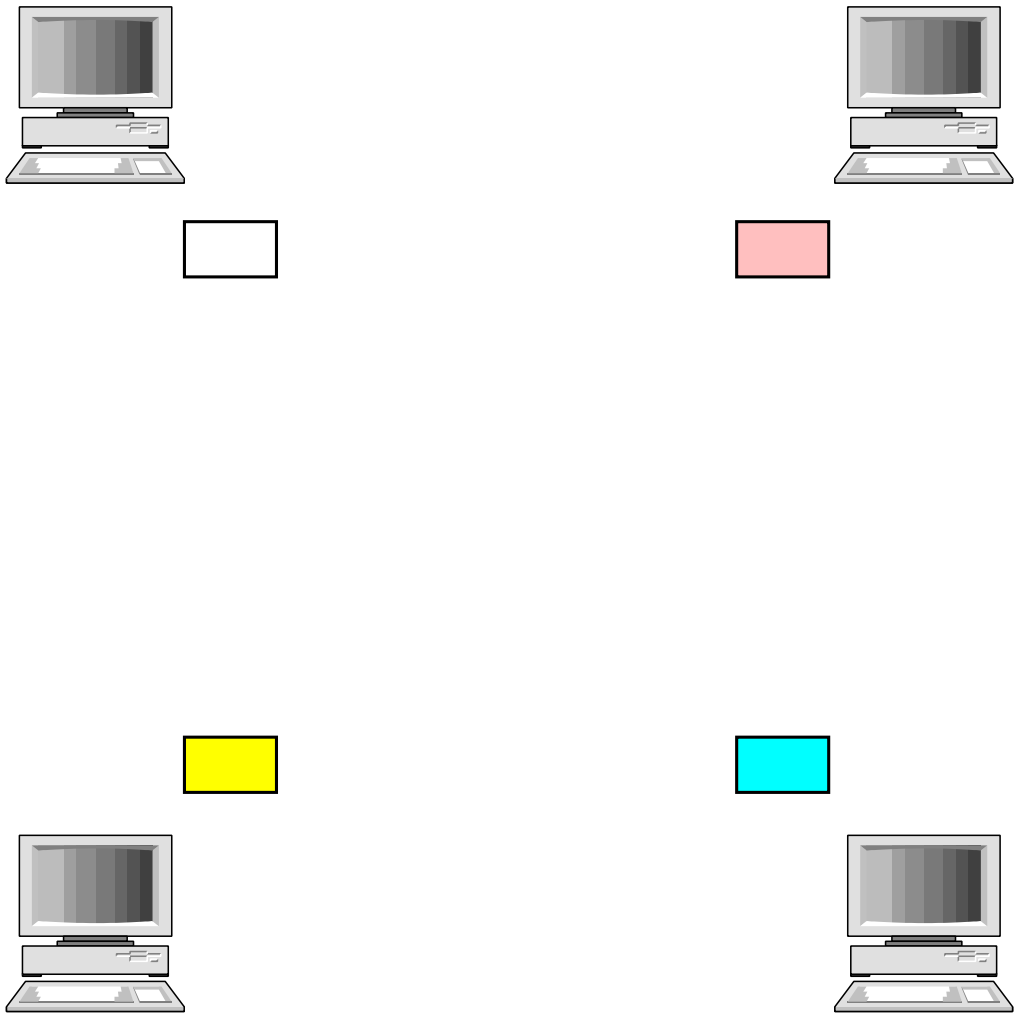
Scatter: Změna struktury dat



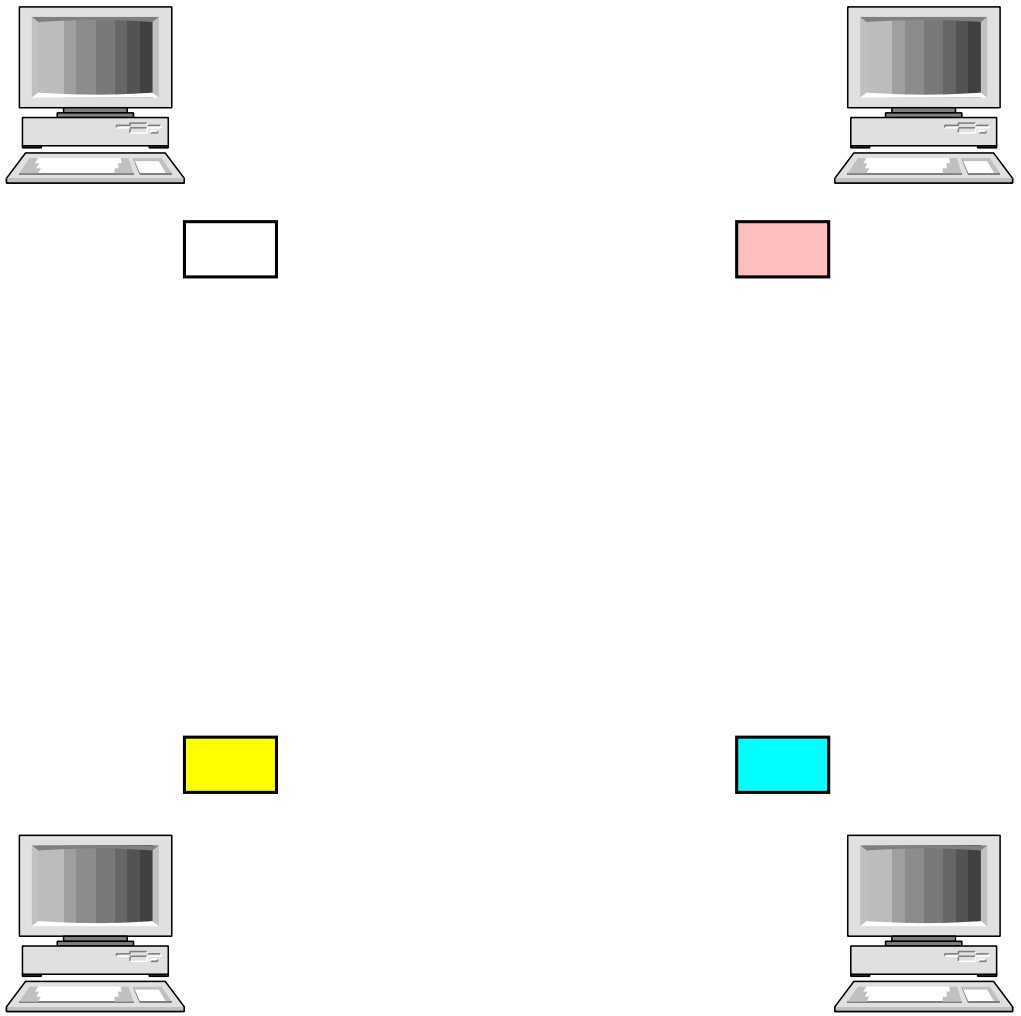
Scatter: Změna struktury dat



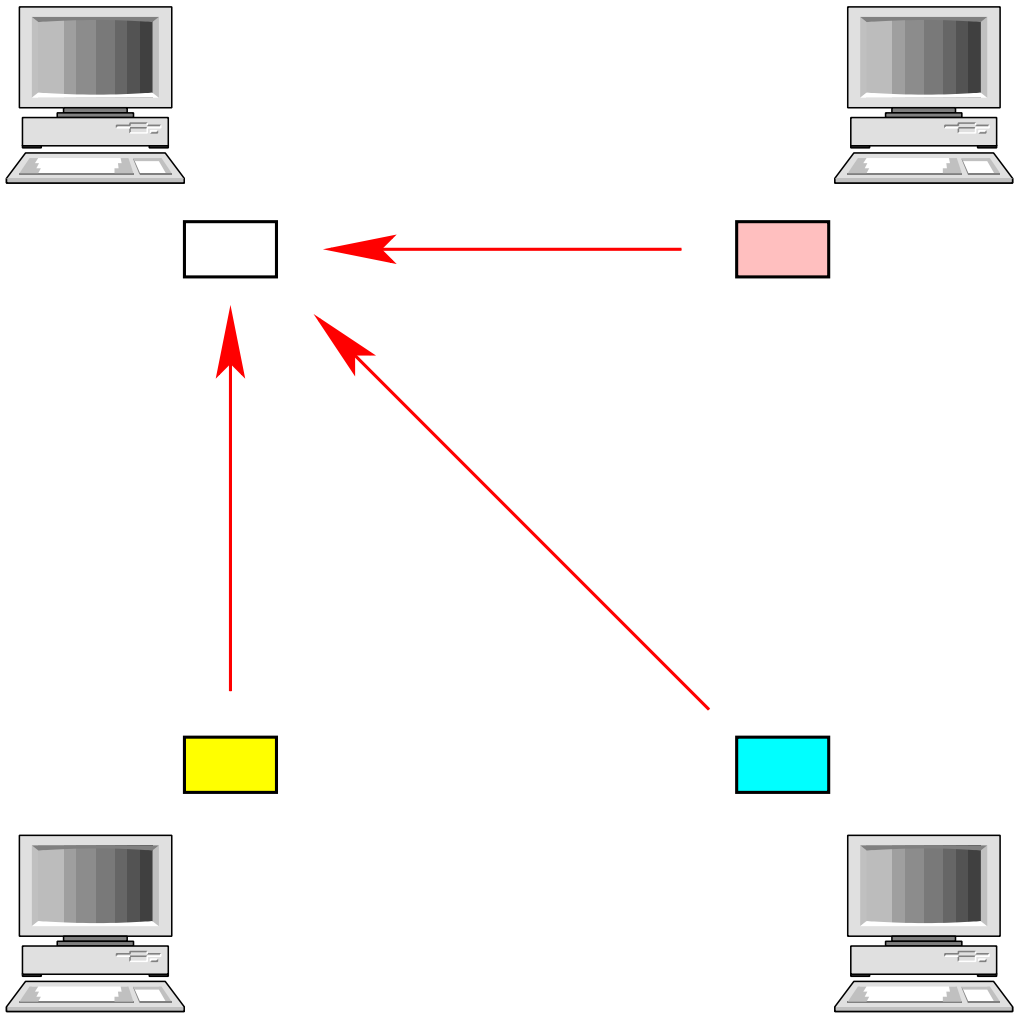
Scatter: Změna struktury dat



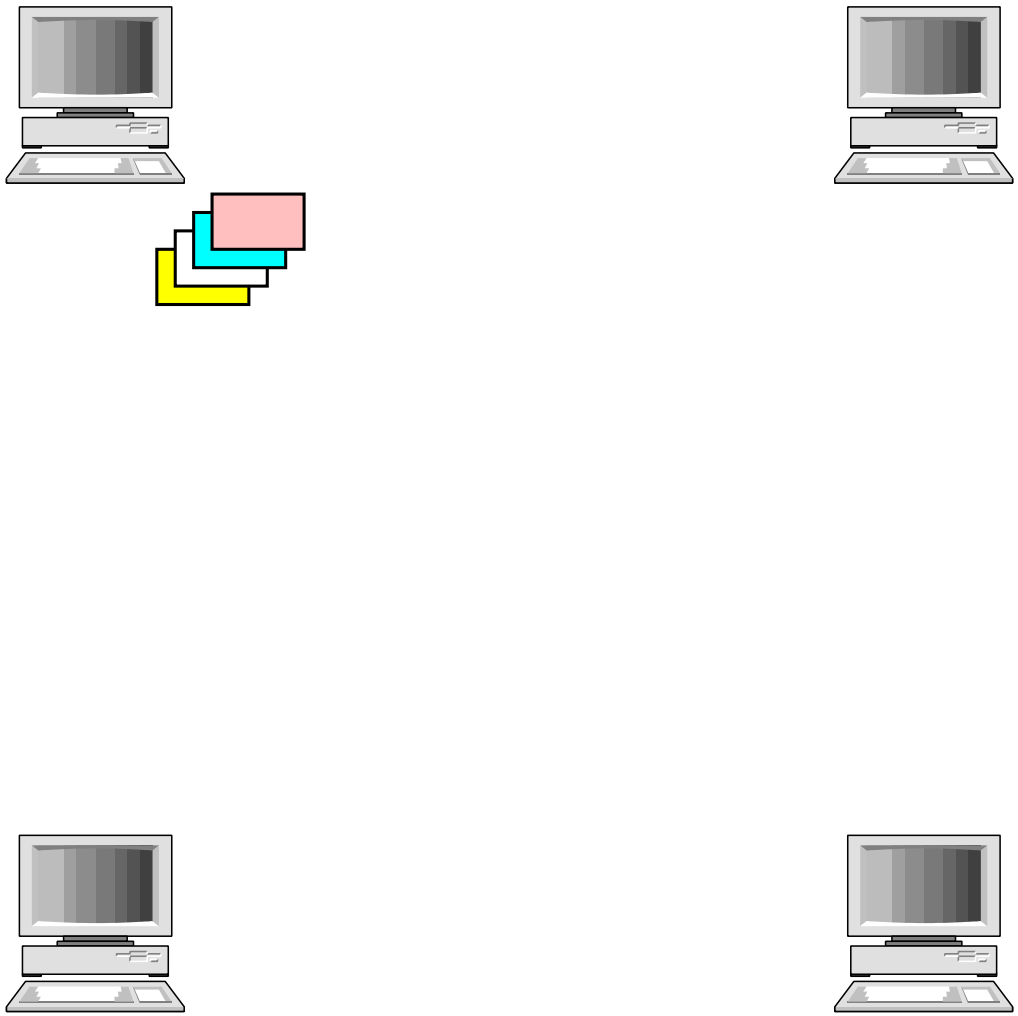
Gatter: Změna struktury dat



Gatter: Změna struktury dat



Gatter: Změna struktury dat



Cyklický posun

Permutace

Permutace

- Obecnější komunikační primitiva
- Současně probíhající OTO přerozdělování dat
- Jedna úloha posílá data jedné jiné úloze

Příklad

- Cyklický posun o q (circular q-shift)
- p úloh
- Úloha i posílá data úloze $(i + q)\ mod\ p$

Použití

- Specifické maticové operace
- Vyhledávání vzorů v textu či obrazu

Cyklický posun – Sítě ve tvaru mřížek

1-dimenzionální

- Intuitivně: rotace o q pozic
- Směr rotace závislý na q, určí se dle výrazu $min(q, p - q)$

Dvourozměrné sítě

- Akcelerace cyklického posunu s využitím druhé dimenze než, ve které probíhá posun
- Jedna dimenze má rozměr $\sqrt{p}$ uzlů
- Posun v druhé dimenzi akceleruje o $\sqrt{p}$ kroků

Cena

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

Příklad

- Síť 4x4 uzly, cyklický posun o 5

Cyklický posun – Síť ve tvaru hyperkostky

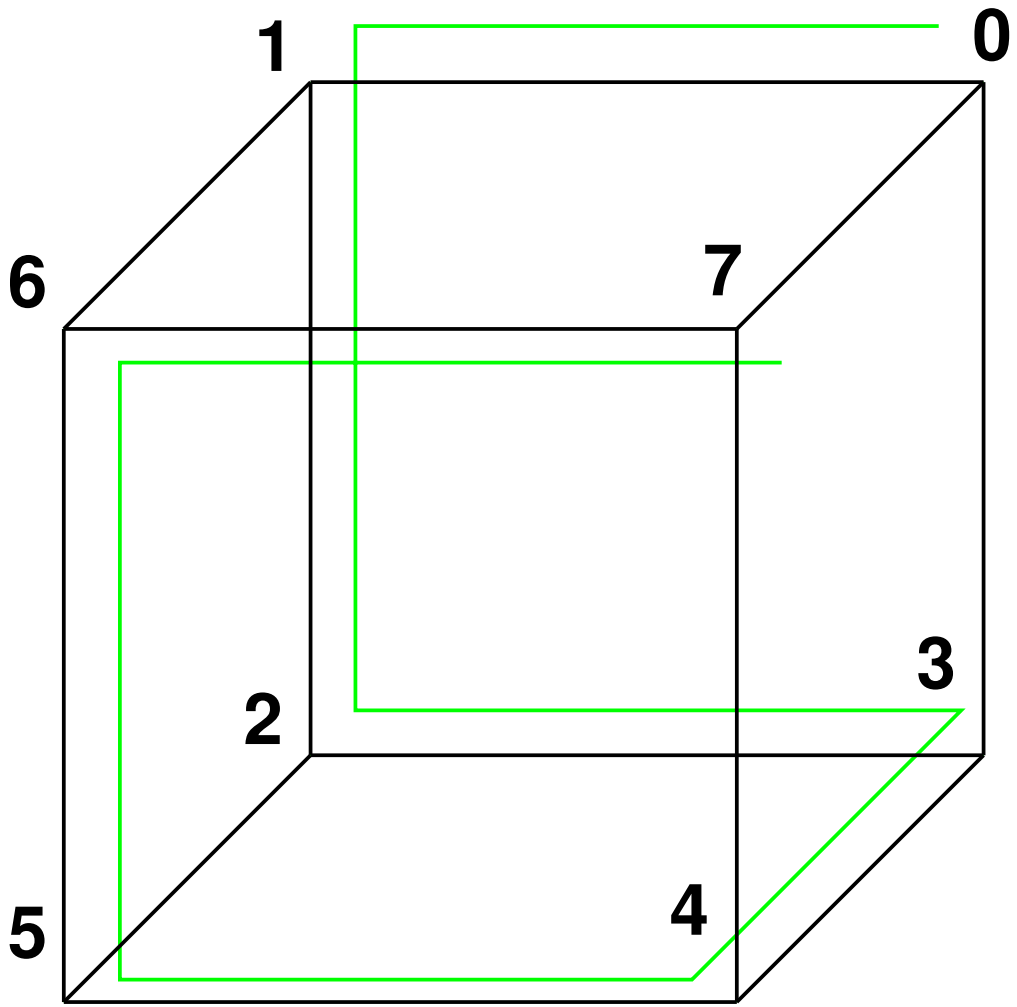
Princip

- Mapování lineárního pole na hyperkostku dimenze d
  - Zrcadlený šedý kód (reflected Grey code): i $= G(i, d)$
  - $G(0, 1) = 0$
  - $G(1, 1) = 1$
  - $G(i, x + 1) =$ if $i < 2^x$ $G(i, x)$ else $2^x + G(2^{x+1} - 1 - i, x)$
- q se vyjádří jako součet mocnin čísla 2
- Posun proběhne v tolika fázích, kolik je členů v součtu

Cena

- Uzly ve vzdálenosti mocniny čísla 2, jsou od sebe vzdáleny nejvýše 2 kroky (vlastnost kódu)
- Členů v součtu je nejvýše $log\ p$
- Komunikace probíhá bez blokování linek
- $T = (t_s + mt_w)(2log\ p)$
- Se zpětným posunem je počet sčítanců max. $(log\ p)/2$
- $T = (t_s + mt_w)(log\ p)$

Binary Reflected Grey code



Práce na doma

Nebodovaný domácí úkol

- Projděte si probraná komunikační primitiva, a znovu odvoďte jejich cenu pro jednotlivé topologie.

IB109 Návrh a implementace paralelních systémů

Programování aplikací pro prostředí s distribuovanou pamětí

Jiří Barnat

Principy programování s předáváním zpráv

Vymezení prostředí

Paradigma – předávání zpráv

- Nesdílený adresový prostor
- Explicitní paralelismus

Pozorování

- Data explicitně dělena a umístěna do jednotlivých lokálních adresových prostorů.
- Datová lokalita je klíčová vlastnost pro výkon.
- Komunikace vyžaduje aktivní účast komunikujících stran.
- Existují efektivně implementované podpůrné knihovny.
- Programátor je zodpovědný za paralelizaci algoritmu.

Struktura programů s předáváním zpráv

Asynchronní paradigma

- Výpočet začne ve stejný okamžik (synchronně), ale ...
- Probíhá asynchronně (různá vlákna různou rychlostí).
- Možnost synchronizace v jednotlivých bodech výpočtu.
- Neplatí "Trojúhelníková nerovnost" v komunikaci.

Další vlastnosti

- Vykazuje nedeterministické chování.
- Těžší prokazování korektnosti.
- Možnost provádění zcela odlišného kódu na jednotlivých procesních jednotkách.
- Typicky však "Single program multiple data".
- Každá procesní jednotka má jednoznačnou identifikaci.

Send a Receive – Základní stavební kameny

Send a Receive

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelemns, int src)
```

- `sendbuf` – ukazatel na bafr (blok paměti), kde jsou umístěna data připravená k odeslání.
- `recvbuf` – ukazatel na bafr (blok paměti), kam budou umístěna přijatá data.
- `nelems` – počet datových jednotek, které budou poslány, či přijaty (délka zprávy).
- `dest` – adresát odesílané zprávy, tj. *ID* toho, komu je zpráva určena.
- `src` – odesilatel zprávy, tj. *ID* toho, kdo zprávu poslal, nebo toho, od koho chci zprávu přijmout.

Příklad na send a receive

Příklad

- Paralelní systém s 2 procesy

```
1      P0                      P1
2
3      a = 100;                receive (&a, 1, 0);
4      send(&a, 1, 1);         printf("%d\n", a);
5      a = 0;
```

Výstup a efektivita

- Co by mělo být na výstupu procesu `P1`?
- Asynchronní sémantika `send()` a `receive()`.
- Nutné z důvodu zachování výkonu aplikace.
- Co může být na výstupu procesu `P1`?

Blokující nebafrované operace

Blokující operace

- Operace `send()` ukončena až tehdy, je-li to bezpečné vzhledem k sémantice, tj. že příjemce obdrží to, co bylo obsahem odesílaného bafru v okamžiku volání operace `send()`.
- Ukončení operace `send()` nevynucuje a negarantuje, že příjemce již zprávu přijal.
- Operace `receive()` skončí po příjetí dat a jejich umístění na správné místo v paměti.

Nebafrované operace

- Operace `send()` skončí až po dokončení operace komunikace, tj. až přijímací proces zprávu přijme.
- V rámci operací `send()` a `receive()` před samotným přenosem dat probíhá synchronizace obou participujících stran (handshake).

Blokující nebafrované operace – problémy

Prodlevy

- Způsobeno synchronizací před vlastní komunikací.
- Proces, který dosáhne bodu, kdy je připraven komunikovat čeká, až do stejného bodu dospěje i druhý proces.
- Volání `send()` a `receive()` ve stejný okamžik nelze garantovat na úrovni kódu.
- Nemá velký vliv, pokud dominuje čas komunikace.

Uváznutí (deadlock)

```
1   P0                          P1
2
3   send(&a, 1, 1);             send(&a, 1, 0);
4   receive(&b, 1, 1);          receive(&b, 1, 0);
```

Blokující bafrované operace

Bafr v bafrované komunikaci

- Extra paměť zdánlivě mimo adresový prostor procesů.
- Mezisklad zpráv při komunikaci.

Bafrované komunikační operace

- Operace `send()` skončí v okamžiku, kdy odesílaná data kompletně překopírována do bafru.
- Případná modifikace posílaných dat po skončení operace `send()`, ale před započetím vlastní komunikace se neprojeví.
- Volání `send()`, vlastní komunikace a následné volání `receive()` na přijímací straně se nemusí časově překrývat.

Blokující bafrované operace – problémy

Režie související s bafrováním

- Eliminace prodlev za cenu režie bafrování.
- Ve vysoce synchroniích aplikacích může být horší než používání blokujících nebafrovaných operací.

Velikost bafrů

- Pokud odesílatel generuje zprávy rychleji, než je příjemce schopen zprávy přijímat, velikost bafrů může neúměrně růst (problém producent-konzument).
- Pokud je velikost pro bafry omezená, může docházet (a to nedeterministicky) k situaci, kdy je předem daná velikost bafrů nedostatečná (buffer overflow).
- Případné samovolné blokování odesílatele do té doby, než odesílatel přijme nějaká data a bafry se uvolní, může vést k uváznutí, podobně jako v případě nebafrované blokující komunikace.

Blokující bafrované operace – problémy

Uváznutí (i bez blokování odesílatele)

```
1   P0                          P1
2
3   receive(&b, 1, 1);          receive(&b, 1, 0);
4   send(&a, 1, 1);             send(&a, 1, 0);
```

Asymetrický model blokující bafrované komunikace

- Neexistence odpovídajících prostředků pro bafrovanou komunikaci na úrovni komunikační vrstvy.
- Operace `send()` je blokující nebafrovaná.
- Přijímací proces je přerušen v běhu a zpráva je přijata do bafru, kde čeká, dokud přijímací proces nezavolá odpovídající operaci `receive()`.
- Dedikované vlákno pro obsluhu komunikace.

Neblokující operace

Motivace

- Komunikace, která nezpůsobuje prodlevy.

Neblokující operace

- Volání funkce může skončit dříve, než je to sémanticky bezpečné.
- Existuje funkce na zjištění stavu komunikující operace.
- Program nesmí modifikovat odesílaná data, dokud komunikace neskončí.
- Po dobu trvání komunikace program může vykonávat kód.
- Překrývání výpočtu a komunikace.

HW Realizace

- DMA

Přehled komunikačních módů

	Blokující	Neblokující
Bafrované		`send` skončí jakmile je inicializován DMA přenos
Nebafrované	`send` skončí až po skončení odpovídajícího `receive`	`send` skončí ihned, odešle se pouze požadavek na komunikaci
	Sémantiku operací nelze porušit	Korektní dokončení operací nutné zjišťovat opakovaným dotazováním se

Message Passing Interface

Message Passing Interface

- Standardizuje syntax a sémantiku komunikačních primitiv
- Přes 120 knihovních funkcí
- Rozhraní pro C, Fortran
- MPI verze 1.2
- MPI verze 2.0 (Paralelní I/O, C++ rozhraní, ...)
- Existují různé implementace standardu
  - mpich
  - LAM/MPI
  - Open MPI
- `http://www.mpi-forum.org/`

Minimální použití MPI

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

- Definice typů a konstant: `#include "mpi.h"`
- Návratová hodnota při úspěšném volání fce: `MPI_SUCCESS`
- Kompilace: `mpicc`, `mpiCC`, `mpiC++`
- Spuštění programu: `mpirun`

Inicializace a ukončení MPI knihovny

Inicializace

- Nastavuje MPI prostředí
- Musí být voláno na všech procesorech
- Musí výt voláno jako první MPI funkce
- `argv` nesmí být modifikováno před voláním `MPI_Init`
- `MPI_Init(int *argc, char ***argv)`

Finalizace

- Ukončuje MPI prostředí
- Musí být voláno na všech procesech
- Nesmí být následováno voláním MPI funkce
- Provádí různé úklidové práce
- `int MPI_Finalize()`

MPI komunikátory

Komunikační domény

- Sdružování participujících procesorů do skupin
- Skupiny se mohou překrývat

Komunikátory

- Proměnné, které uchovávají komunikační domény
- Typ `MPI_Comm`
- Jsou argumentem všech komunikačních funkcí MPI
- Výchozí komunikátor: `MPI_COMM_WORLD`

Zjišťování velikosti domény a identifikátoru v rámci domény

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `rank` je identifikátor procesu v dané doméně
- `rank` číslo v intervalu `[0,size-1]`

Hello World

```
1   #include "mpi.h"
2
3   main (int argc, char *argv[])
4   {
5     int npes, myrank;
6
7     MPI_Init (&argc, &argv);
8     MPI_Comm_size(MPI_COMM_WORLD, &npes);
9     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10
11    printf("Hello World! (%d out of %d)",
12           myrank,npes);
13
14    MPI_Finalize();
15  }
```

Posílání zpráv

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

- Odešle data odkazovaná ukazatelem `buf`
- Na data se nahlíží jako na sekvenci instancí typu `datatype`
- Odešle se `count` po sobě jdoucích instancí
- `dest` je rank adresáta v komunikační doméně určené komunikátorem `comm`
- `tag`
  - Přiložená informace typu `int` v intervalu `[0,MPI_TAG_UB]`
  - Pro příjemce viditelná bez čtení obsahu zprávy
  - Typicky odlišuje typ zprávy

Korespondence datových typů MPI a C

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

Přijímání zpráv

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

- Přijme zprávu od odesílatele s rankem `source` v komunikační doméně `comm` s tagem `tag`
- `source` může být `MPI_ANY_SOURCE`
- `tag` může být `MPI_ANY_TAG`
- Zpráva uložena na adrese určené ukazatelem `buf`
- Velikost bafru je určena hodnotami `datatype` a `count`
- Pokud je bafr malý, návratová hodnota bude `MPI_ERR_TRUNCATE`

Přijímání zpráv – MPI_Status

`MPI_Status`

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
};
```

- Vhodné zejména v případě příjmu v režimu `MPI_ANY_SOURCE` nebo `MPI_ANY_TAG`
- `MPI_Status` drží i další informace, například skutečný počet přijatých dat (délka zprávy)

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

Posílání a přijímání zpráv – sémantika

`MPI_Recv`

- Volání skončí až jsou data umístěna v bafru
- Blokující receive operace

`MPI_Send`

- MPI standard připouští 2 různé sémantiky
  1) Volání skončí až po dokončení odpovídající receive operace
  2) Volání skončí jakmile jsou posílaná data zkopírována do bafru
- Změna odesílaných dat je vždy sémanticky bezpečná
- Blokující send operace

Možné důvody uváznutí

- Jiné pořadí zpráv při odesílání a přijímání
- Cyklické posílání a přijímání zpráv (večeřící filozofové)

Současné posílání a přijímání zpráv

`MPI_Sendrecv`

- Operace pro současné přijímání i odesílání zpráv
- Nenastává cyklické uváznutí
- Bafry pro odesílaná a přijímaná data musí být různé

```
int MPI_Sendrecv (
  void *sendbuf, int sendcount,
  MPI_Datatype senddatatype, int dest, int sendtag,
  void *recvbuf, int recvcount,
  MPI_Datatype recvdatatype,
  int source, int recvtag, MPI_Comm comm,
  MPI_Status *status)
```

Současné posílání a přijímání zpráv – Sendrecv_replace

Problémy `MPI_Sendrecv`

- Bafry pro odesílaná a přijímaná data zabírají 2x tolik místa
- Složitá manipulace s daty díky 2 různým bafrům

`MPI_Sendrecv_replace`

- Operace odešle data z bafru a na jejich místo nakopíruje přijatá data

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

Neblokující komunikace

Neblokující Send a Receive

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

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

`MPI_Request`

- Identifikátor neblokující komunikační operace
- Potřeba při dotazování se na dokončení operace

Dokončení neblokujících komunikačních operací

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

- Nulový `flag` znamená, že operace ještě probíhá
- Při prvním volání po dokončení operace se naplní `status`, zničí `request` a `flag` nastaví na *true*

```
int MPI_Wait(MPI_Request *request,
             MPI_Status *status)
```

- Blokující čekání na dokončení operace
- Po dokončení je `request` zničen a `status` naplněn

```
int MPI_Request_free(MPI_Request *request)
```

- Explicitní zničení objektu `request`
- Nemá vliv na probíhající operaci

Neblokující operace – poznámky

Párování

- Neblokující a blokující send a receive se mohou libovolně kombinovat

Uváznutí

- Neblokující operace řeší většinu problémů s uváznutím
- Neblokující operace mají vyšší paměťové nároky
- Později zahájená neblokující operace může skončit dříve

Kolektivní komunikace

Kolektivní operace

- Množina participujících procesů je určena komunikační doménou (`MPI_Comm`).
- **Všechny procesy v doméně musí volat odpovídající `MPI` funkci.**
- Obecně se kolektivní operace nechovají jako bariéry, tj. jeden proces může dokončit volání funkce dříve, než jiný proces vůbec dosáhne místa volání kolektivní operace.
- Forma virtuální synchronizace.
- Nepoužívají tagy (všichni vědí jaká operace se provádí).
- Pokud je nutné specifikovat zdrojový, či cílový proces, musí tak učinit všechny participující procesy a jejich volba cílového, či zdrojového procesu musí být shodná.
- `MPI` podporuje dvě varianty kolektivních operací
  - posílají se stejně velká data (např. `MPI_Scatter`)
  - posílají se různě velká data (např. `MPI_Scatterv`)

Bariéra

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

- Základní synchronizační primitivum.
- Volání funkce skončí pokud všechny participující procesy zavolají `MPI_Barrier`.
- Není nutné, aby volaná funkce byla "na stejném místě v programu".

Kolektivní komunikační primitiva a MPI

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

Všesměrové vysílání – OTA

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

- Rozesílá data uložená v bafru `buf` procesu `source` ostatním procesům v doméně `comm`.
- Kromě `buf` musí být parametry funkce shodné ve všech participujících procesech.
- Parametr `buf` na ostatních procesech slouží pro identifikaci bafru pro příjem dat.

Redukce

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

- Data ze `sendbuf` zkombinována operací `op` do `recvbuf` procesu `target`.
- Všichni participující musí poskytnout `recvbuf` i když výsledek uložen pouze na procesu `target`.
- Hodnoty `count`, `datatype`, `op`, `target` musí být shodné ve všech volajících procesech.
- Možnost definovat vlastní operace typu `MPI_Op`.

Operátory redukce

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

Datové páry a Kompletní Redukce

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

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

Prefixový součet

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

- Provádí prefixovou redukci.
- Proces s rankem *i* má ve výsledku hodnotu vzniklou redukcí hodnot procesů s rankem 0 až *i* včetně.
- Jinak shodné s redukcí.

Gather

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

- Všichni posílají stejný typ dat.
- Cílový proces obdrží p bafrů seřazených dle ranku odesílatele.
- `recvbuf`, `recvcount`, `recvbuf` platné pouze pro proces s rankem `target`.
- `recvcount` je počet odeslaných dat jedním procesem, nikoliv celkový počet přijímaných dat.

Allgather

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

- Bez určení cílového procesu, výsledek obdrží všichni.
- `recvbuf`, `recvcount`, `recvdatattype` musí být platné pro všechny volající procesy.

Vektorová varianta Gather

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

- Odesílatelé mohou odesílat různě velká data (různé hodnoty `sendcount`).
- Pole `recvcounts` udává, kolik dat bylo přijato od jednotlivých procesů.
- Pole `displs` udává, kde v bafru `recvbuf` začínají data od jednotlivých procesů.

Vektorová varianta Allgather

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

Scatter

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

- Posílá různá data **stejné** velikosti všem procesům.

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

- Posílá různá data **různé** velikosti všem procesům.

Alltoall

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

- Posílá stejně velké části bafru `sendbuf` jednotlivým procesům v doméně `comm`.
- Proces *i* obdrží část velikosti `sendcount`, která začíná na pozici `sendcount * i`.
- Každý proces má v bafru `recvbuf` na pozici `recvcount * i` data velikosti `recvcount` od procesu *i*.

Vektorová varianta Alltoall

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

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

Skupiny, komunikátory a topologie

Dělení komunikační domény

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

- Kolektivní operace, musí být volána všemi.
- Parametr `color` určuje výslednou skupinu/doménu.
- Parametr `key` určuje rank ve výsledné skupině.
  - Při shodě `key` rozhoduje původní rank.

Mapování procesů

Nevýhody "ručního"mapování

- Pravidla mapování určena v době kompilace programu.
- Nemusí odpovídat optimálnímu mapování.
- Nevhodné zejména v případech nehomogenního prostředí.

Mapování přes MPI

- Mapování určeno za běhu programu.
- MPI knihovna má k dispozici (alespoň částečnou) informaci o síťovém prostředí (například počet použitých procesorů v jednotlivých participujících uzlech).
- Mapování navrženo s ohledem na minimalizaci ceny komunikace.

Kartézské topologie

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

- Pokud je v původní doméně `comm_old` dostatečný počet procesorů, tak vytvoří novou doménu `comm_cart` s virtuální kartézskou topologií.
- Funkci musí zavolat všechny procesy z domény `comm_old`.
- Parametry kartézské topologie
  - `ndims` – počet dimenzí
  - `dims[]` – pole rozměrů jednotlivých dimenzích
  - `periods[]` – pole příznaků cyklické uzavřenosti dimenzí
- Příznak `reorder` značí, že ranky procesů se mají v rámci nové domény vhodně přeuspořádat.
- Nepoužité procesy označeny rankem `MPI_COMM_NULL`.

Koordináty procesorů v kartézských topologiích

- Komunikační primitiva vyžadují rank adresáta.
- Překlad z koordinátů (`coords[]`) do ranku

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

- Překlad z ranku na koordináty
- `maxdims` je velikost vstupního pole `coords[]`

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

Dělení kartézských topologií

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

- Pro dělení kartézských topologií na topologie s menší dimenzí.
- Pole příznaků `keep_dims` určuje, zda bude odpovídající dimenze zachována v novém dělení.

Příklad

- Topologie o rozměrech $2 \times 4 \times 7$.
- Hodnotou `keep_dims` = `{true,false,true}`.
- Vzniknou 4 nové domény o rozměrech 2×7.

Případová studie implementace verifikačního nástroje DiVinE

DiVinE-cluster

DiVinE-Cluster

- Softwarový nástroj pro verifikaci protokolů (LTL MC).
- Problém detekce akceptujícího cyklu v grafu.
- Algoritmy paralelně prochází graf konečného automatu.
- Standardní průzkumová dekompozice.

Algoritmus MAP

- Detekuje, zda existuje akceptující vrchol, který je svůj vlastní předchůdce.

Algoritmus OWCTY

- Označuje vrcholy, které nejsou součástí akceptujícího cyklu
  - nemají přímé předchůdce (neleží na cyklu),
  - nemají akceptující předky (neleží na akceptujícím cyklu).

Co bylo špatně

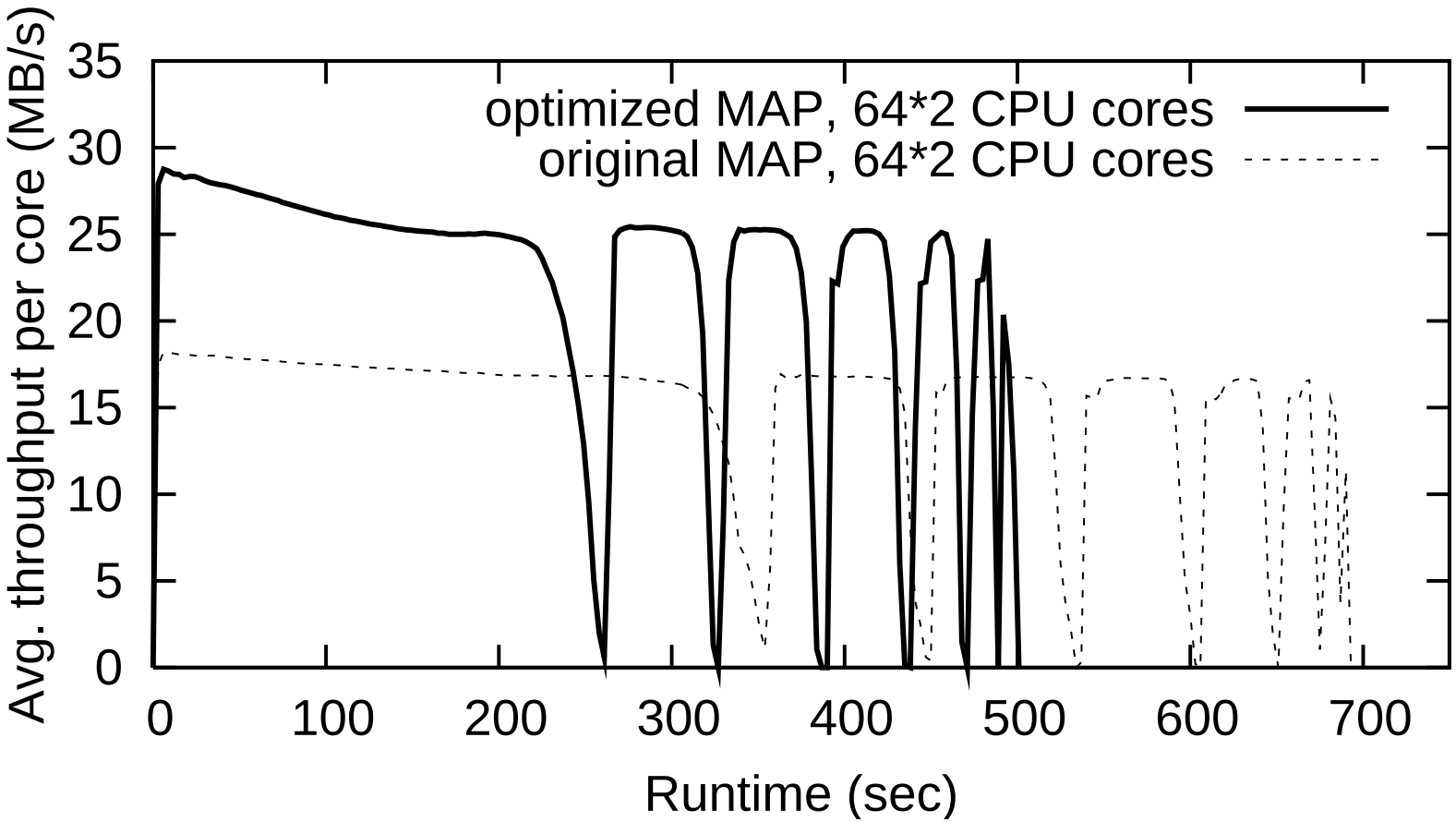
Obecně

- Při testování na standardním Ethernetu, na malém počtu počítačů, některé nedokonalosti zůstaly neodhaleny.

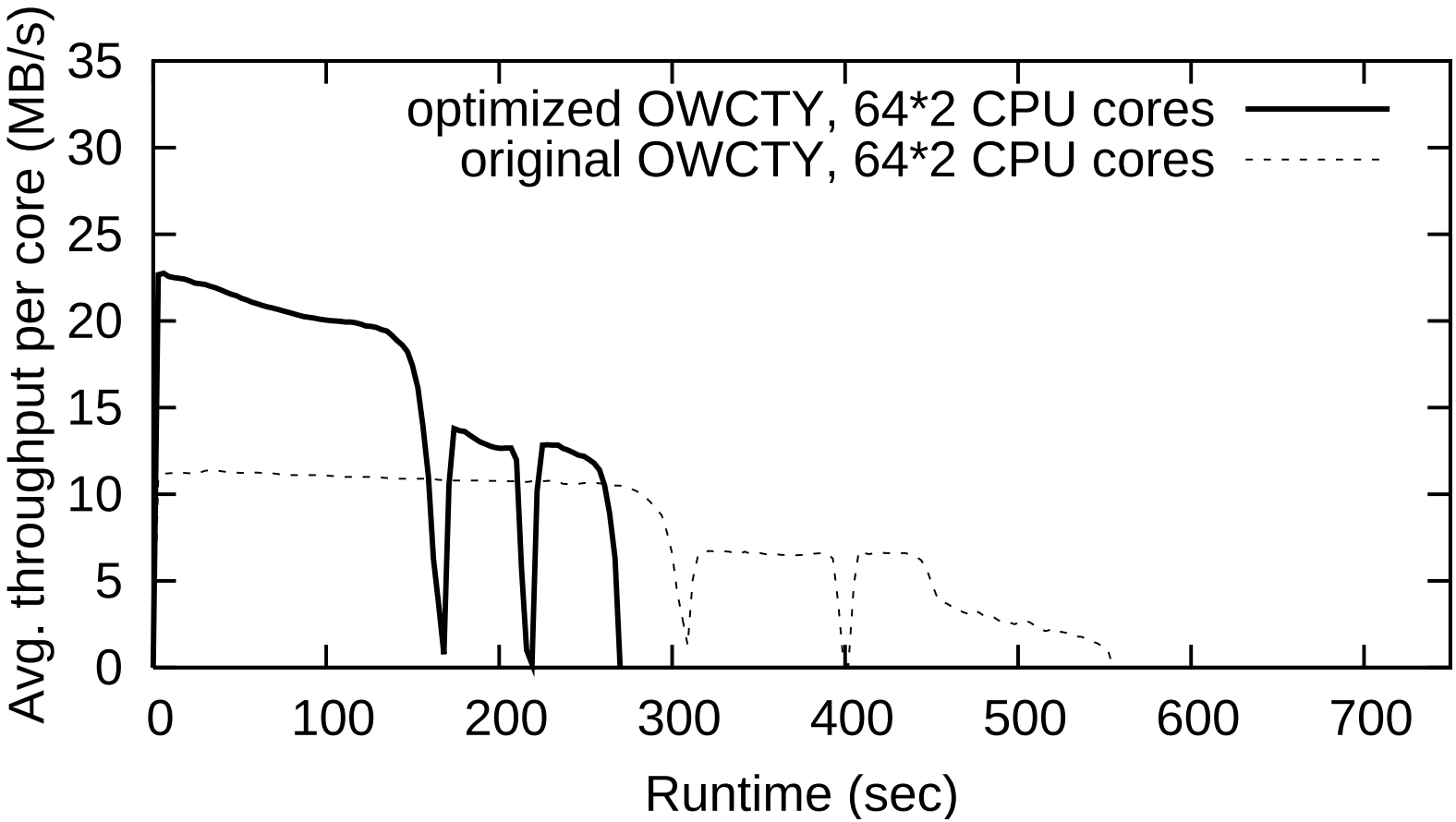
Konkrétní problémy

- Po inicializaci některých počítačů, aktivita těchto počítačů zabránila inicializaci ostatních počítačů – obrovské prodlevy při startu výpočtu.
- Kombinace posílaných zpráv (bafrování na aplikační úrovni)
  - Příliš časté vylívání – komunikovaly se prázdné bafry.
  - Vylití všech bufferů najednou – náhlá velká zátěž sítě (contention).
  - Používání časových známek a vylívání na základě času – netriviální režie způsobená ověřováním zda uplynulo dané množství času.
- Příliš časté dotazování se na příchozí zprávy.

Výsledky optimalizací



Výsledky optimalizací



Paralelní výkon nástroje DiVinE

Nodes	Total cores	Runtime (s)		Efficiency	
		MAP	OWCTY	MAP	OWCTY
1	1	956.8	628.8	100%	100%
16	16	73.9	42.5	81%	92%
16	32	39.4	22.5	76%	87%
16	64	20.6	11.4	73%	86%
64	64	19.5	10.9	77%	90%
64	128	10.8	6.0	69%	82%
64	256	7.4	4.3	51%	57%

Tabulka: Efficiency of MAP and OWCTY

Domácí úkol (4 body)

Zadání

- Napište MPI aplikaci, ve které si participující procesy mezi sebou zvolí jednoho velitele.
- Volba musí být realizována pomocí generování náhodných čísel a jejich výměny mezi jednotlivými MPI procesy.
- Při spuštění tato aplikace vypíše autorovo UČO.
- Kód spustitelný a přeložitelný na `nymfe50`.

Odevzdání Odevzdání

- Termín do 11. 5. 2016 23:59.
- Odevzdávárna v ISu, zabaleno programem TAR a komprimováno GZIPem: *IB109_02_učo.tar.gz*
- Archiv obsahuje sbalený adresář IB109_02_učo.
- Povinně obsahuje Makefile.
- Provedení `make` uvnitř adresáře přeloží a spustí aplikaci.
- Nesprávné odevzdání jde na vrub studenta.

IB109 Návrh a implementace paralelních systémů

Analytický model paralelních programů

Jiří Barnat

Analytický model paralelních programů

Vyhodnocování sekvenčních algoritmů

- Časová a prostorová složitost
- Funkce velikosti vstupu

Vyhodnocování paralelních algoritmů

- Paralelní systém = algoritmus + platforma
- Složitost
- Přínos paralelismu
- Přenositelnost
- ...

V této kapitole

- Jak posuzovat výkon paralelních algoritmů a systémů

Režie (overhead) paralelních programů

Režie (overhead) paralelních programů

N-násobné zrychlení

- S použitím n-násobných HW zdrojů, lze očekávat n-násobné zrychlení výpočtu.
- Nastává zřídka z důvodů režií paralelního řešení.
- Pokud nastane, je jeho důvodem často neoptimální řešení sekvenčního algoritmu.

Důvody

- Režie – interakce, prostoje, složitost paralelního algoritmu
- Nerovnoměrné zvyšování HW zdrojů
  - Zvýšení počtu procesorů nepomůže, pokud aplikace není závislá na čistém výpočetním výkonu

Režie (overhead) paralelních programů

Komunikace

- Cena samotné komunikace.
- Příprava dat k odeslání, zpracování přijatých dat.

Prostoje (Lelkování, Idling)

- Nerovnoměrná zátěž na jednotlivá výpočetní jádra.
- Čekání na zdroje či data.
- Nutnost synchronizace asynchronních výpočtů.

Větší výpočetní složitost paralelního algoritmu

- Sekvenční algoritmus nejde paralelizovat (DFS postorder).
- Typicky existuje vícero paralelních algoritmů, je nutné charakterizovat, čím se platí za paralelismus.

Metriky výkonosti

Čas výpočtu (Execution time)

Otázky

- Jak měřit výkon/kvalitu paralelního algoritmu?
- Podle čeho určit nejlepší/nejvhodnější algoritmus pro danou platformu?

Čas výpočtu – sekvenční algoritmus

- Doba, která uplyne od spuštění výpočtu do jeho ukončení.
- T_S

Čas výpočtu – paralelní algoritmus

- Doba, která uplyne od spuštění výpočtu do doby, kdy skončí poslední z paralelních výpočtů.
- Zahrnuje distribuci vstupních i sběr výstupních dat.
- T_P

Celková režie paralelismu (Total overhead)

Celková režie:

- Označujeme T_O
- Veškeré elementy způsobující režii paralelního řešení.
- Celkový čas paralelního výpočtu bez času potřebného pro výpočet problému optimálním sekvenčním řešením.
- Paralelní a sekvenční algoritmy mohou být zcela odlišné.
- Sekvenčních algoritmů může existovat víc.
- T_S čas výpočtu nejlepšího sekvenčního algoritmu (řešení).

Funkce celkové režie

- Jaký je čas výpočtu jednotlivých procesů?
- Doba od skončení výpočtu jednoho procesu do skončení celého paralelního výpočtu se považuje za režii (idling).
- $T_O = pT_P - T_S$

Zrychlení (Speed-up)

Základní přínos paralelizace

- Problémy jdou vždy řešit sekvenčním algoritmem.
- Paralelizací lze dosáhnout pouze zrychlení výpočtu.
- Ostatní výhody jsou diskutabilní a těžko měřitelné.

Základní míra účinnosti samotné paralelizace

- Poměr časů potřebných k vyřešení úlohy na jedné procesní jednotce a p procesních jednotkách.
- Uvažuje se vždy čas nejlepšího sekvenční algoritmu.
- V praxi se často (a nesprávně) používá čas potřebný k vyřešení úlohy paralelním algoritmem spuštěném na jedné procesní jednotce.
- $S = \frac{T_S}{T_P}$.

Zrychlení

Teoretická hranice zrychlení

- S použitím p procesních jednotek je maximální zrychlení p.

Super-lineární zrychlení

- Jev, kdy zrychlení je větší jak p.

Pozorování

- Zrychlení lze měřit i asymptoticky.

Příklad – sčítání n čísel s použitím n procesních jednotek.

- Sekvenčně je potřeba $\Theta(n)$ operací, čas $\Theta(n)$
- Paralelně je potřeba $\Theta(n)$ operací, ale v čase $\Theta(log\ n)$
- Zrychlení $S = \Theta(\frac{n}{log\ n})$

Super-lineární zrychlení

Falešné super-lineární zrychlení

- Uvažme datovou distribuci na 2 procesní jednotky, a operaci kvadratickou ve velikosti dat.
- Zrychlení $S = \frac{n^2}{(\frac{n}{2})^2} = 4$ při použití 2 procesních jednotek.
- Problém – neuvažován optimální sekvenční algoritmus.

Skutečné super-lineární zrychlení

- Větší agregovaná velikost cache pamětí.
- Při snížení množství dat na jeden procesní uzel se účinnost cache pamětí zvýší.
- Výpočet je na $\frac{1}{p}$ datech víc jak p-krát rychlejší.

Super-lineární zrychlení v závislosti na instanci problému

- Průzkumová dekompozice úlohy při hledání řešení.
- Paralelní verze může vykonat menší množství práce.
- V konkrétní instanci lze paralelní prohledávání simulovat i sekvenčním algoritmem, obecně ale nelze.

Amdahlův zákon

Otázka

- Jaké je největší možné zrychlení systému, pokud se paralelizací urychlí pouze část výpočtu?

Amdahlův zákon

- $S_{max} = \frac{1}{(1-P)+\frac{P}{S_P}}$
- P – podíl systému, který je urychlen paralelizací
- S_P – zrychlení dosažené paralelizací nad daným podílem

Příklad

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

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

Efektivita

Fakta

- Pouze ideální paralelní systém s p procesními jednotkami může dosahovat zrychlení p.
- Část výpočtu prováděná na jednom procesoru je spotřebována režií. Procesor nevěnuje 100% výkonu řešení problému.

Efektivita

- Lze definovat jako podíl zrychlení S vůči počtu jednotek p.
- $E = \frac{S}{p} = \frac{T_s/T_p}{p} = \frac{T_s}{pT_p}$.
- Zrychlení je v praxi $< p$, efektivita v rozmezí $(0, 1\rangle$.
- "*Podíl času, po který jednotka vykonává užitečný kód.*"

Efektivita – příklady

Příklad 1

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

Příklad 2

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

Cena

Cena řešení problému na daném paralelním systému

- Součin počtu procesních jednotek a doby paralelního výpočtu: $C = p \times T_P$
- Označována také jako "množství práce", které paralelní systém vykoná, nebo jako "pT_p produkt".
- Alternativně lze použít pro výpočet účinnosti ($E = \frac{T_S}{C}$).

Cenově optimální paralelní systém

- Cena sekvenčního výpočtu odpovídá nejlepšímu T_S.
- Paralelní systém je cenově optimální, pokud cena řešení roste asymptoticky stejně rychle jako cena sekvenčního výpočtu.
- Cenově optimální systém musí mít efektivitu rovnou $\Theta(1)$.

Příklad

- Sčítání n čísel na n procesorech
- $C = \Theta(n \log n)$, není cenově optimální

Cena – příklad

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

Vliv granularity na cenovou optimalitu

Vliv granularity na cenovou optimalitu

Tvrzení

- Volbou granularity lze ovlivnit cenu paralelního řešení.

Pozorování

- V praxi $p << n$, přesto navrhujeme algoritmy tak, aby granularita byla až na úrovni jednotlivých položek vstupu, tj. předpokládáme $p = n$.

Deškálování (scale-down)

- Uměle snižujeme granularitu (vytváříme hrubší úlohy).
- Snižujeme overhead spojený s komunikací.
- Může ovlivnit cenu a cenovou optimalitu.

Vliv granularity na cenovou optimalitu

Příklad 1)

- Sčítání n čísel na p procesorech ($n = 16$, $p = 4$)
- Mapování ($i \ mod \ p$), tj. (n/p) čísel na 1 procesor.
- Simulace prvních $log\ p$ kroků stojí $(n/p)log\ p$ operací, tj. proběhne v čase $\Theta((n/p)log\ p)$.
- Následně simulace původního algoritmu probíhá lokálně v paměti jednoho procesoru, tj. v čase $\Theta(n/p)$.
- Celkový T_P je $\Theta((n/p)log\ p)$
- Cena $C = \Theta(nlog\ p)$, tj. není cenově optimální
- Původní T_P bylo $\Theta(log\ n)$, změna o faktor $\frac{n}{p}(\frac{log\ p}{\log\ n})$

Vliv granularity na cenovou optimalitu

Příklad 2)

- Sčítání n čísel na p procesorech ($n = 16$, $p = 4$)
- Nechť n a p jsou mocniny dvojky (n a p jsou soudělné)
- Mapování ($i \ mod \ p$)
- Každá jednotka nejprve sečte data lokálně v čase $\Theta(n/p)$
- Problém redukován na sčítání p čísel na p procesorech
- Celkový T_P je $\Theta(n/p + log\ p)$
- Cena $C = \Theta(n + plog\ p)$,
- Cenově optimální, pokud $n > p\ log\ p$ (cena $C = \Theta(n)$)

Škálovatelnost paralelních programů

Škálovatelnost paralelních programů

Fakta

- Programy jsou testovány na malých vstupních datech na systémech s malým počtem procesních jednotek.
- Použití deškálování zkresluje měření výkonu aplikace.
- Algoritmus, který vykazuje nejlepší výkon na testovaných datech, se může ukázat být tím nejhorším algoritmem, při použití na skutečných datech.
- Odhad výkonu aplikace nad reálnými vstupními daty a větším počtu procesních jednotek je komplikovaný.

Škálovatelnost

- Zachování výkonu a efektivity při zvyšování počtu procesních jednotek a zvětšování objemu vstupních dat.

Závěr

- Je třeba uvážit analytické techniky pro vyhodnocování výkonu a škálování.

Charakteristiky škálování – Efektivita

Efektivita paralelních programů:

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

Celková režie (T_O)

- Přítomnost sekvenční části kódu je nevyhnutelná. Pro její vykonání je třeba čas t_{serial}. Po tuto dobu jsou ostatní procesní jednotky nevyužité.
- $T_O > (p-1)t_{serial}$
- T_O roste minimálně lineárně vzhledem k p, často však i asymptoticky více.

Úloha konstantní velikosti (T_S fixní)

- Se zvyšujícím se p, efektivita nevyhnutelně klesá.
- **Nevyhnutelný úděl všech paralelních programů.**

Charakteristiky škálování – Příklad

Problém

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

Charakteristiky řešení

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

Charakteristiky škálování – Škálovatelný systém

Při zachování T_S

- Rostoucí počet procesních jednotek snižuje efektivitu.

Při zachování počtu procesních jednotek

- Rostoucí T_S (velikost vstupních dat) má tendenci zvyšovat efektivitu.

Škálovatelné systémy

- Efektivitu lze zachovat při souběžném zvyšování T_S a p.
- Zajímáme se o poměr, ve kterém se T_S a p musí zvyšovat, aby se zachovala efektivita.
- Čím menší poměr T_S/p tím lepší škálovatelnost.

Izoefektivita jako metrika škálování

Vztahy

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

W – velikost vstupních dat

Konstantní efektivita

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

Funkce izoefektivity (stejné efektivity)

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

Izoefektivita jako metrika škálování

Funkce izoefektivity

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

Pozorování

- Při konstantní efektivitě, lze T_S vyjádřit jako funkci p
- Vyjadřuje vztah, jak se musí zvýšit T_S při zvýšení p
- Nižší funkce říká, že systém je snáze škálovatelný
- Isoefektivitu nelze měřit u systémů, které neškálují

Izoefektivita jako metrika škálování – příklad

Příklad

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

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

Cenová optimalita a izoefektivita

Optimální cena

- Pro cenově optimální systémy požadujeme $pT_P = \Theta(T_S)$.
- Po dosazení ze základních izo vztahů dostáváme:
  $T_S + T_O(W, p) = \Theta(T_S)$

Cenová optimalita může být zachována pouze pokud:

- režie je nejvýše lineární vůči složitosti sekvenčního algoritmu, tj. funkce $T_O(W, p) \in \mathcal{O}(T_S)$
- složitost sekvenčního algoritmu je minimálně tak velká jako režie, tj. funkce $T_S \in \Omega(T_O(W, p))$

Spodní odhad na izoefektivitu

Izoefektivita

- Čím nižší/menší funkce, tím lepší škálovatelnost.
- Snaha o minimální hodnotu izofunkce.

Sublineární izoefektivita

- Pro problém tvořený N jednotkami práce je optimální cena dosažitelná pouze pro maximálně N procesních jednotek.
- Přidáváním dalších jednotek vyústí v idling některých jednotek a snižování efektivity.
- Aby se efektivita nesnižovala musí množství práce růst minimálně lineárně vzhledem k p.
- Funkce izoefektivity nemůže být sublineární.

Minimální doba výpočtu

Otázka

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

Pozorování

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

Jak zjistit T_P^{min}?

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

Příklad – součet n čísel p procesními jednotkami

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

Asymptotická analýza paralelních programů

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

Otázky

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

Závěr

Návrh paralelních algoritmů ač složitostně neoptimálních je nedílnou a důležitou částí práce programátora paralelních aplikací.

Shrnutí

Co jsme se v kurzu dozvěděli

Paralelní HW platformy

- Principy fungování HW systémů se sdílenou pamětí.
- Komunikace v systémech s distribuovanou pamětí.

Nástroje paralelního programování

- Standardy a knihovny pro implementaci paralelních aplikací.

  `POSIX Threads, OpenMP, MPI (OpenMPI)`
- Principy fungování těchto knihoven

  `Lock-Free datové struktury, Kolektivní komunikace`

Algoritmizace

- Principy návrhu paralelních algoritmů.
- Analytické posuzování paralelních řešení.

IB109 Návrh a implementace paralelních systémů

Paralelizace grafových algoritmů

Jiří Barnat

Grafové algoritmy

Grafové pojmy a reprezentace grafu

Základní pojmy

- Neorientovaný/orientovaný graf
- Hrany, vrcholy, incidentní vrcholy hrany, sousednost
- Cesta, dosažitelnost, cyklus, acyklický graf
- Souvislost, podgraf, kompletní graf, les, strom, DAG
- Váhy na hranách, minimální kostra
- Řídké a husté grafy

Reprezentace grafu

- Explicitní: Matice sousednosti a vah
- Explicitní: Seznam následníků (řídké grafy)
- Implicitní: Funkce pro enumeraci následníků

Minimální kostra: Primův Algoritmus

proc *PRIM_MIN_KOSTRA*(V, E, w, r)
  $W := \{r\}$
  $d[r] := 0$
  forall $v \in (V - W)$
    if $(r, v) \in E$
      then $d[v] := w(r, v)$
      else $d[v] := \infty$
    fi
  while $W \neq V$ do
    vyber u tak, aby $d[u] = min\{d[v] \mid v \in V - W\}$
    $W := W \cup \{u\}$
    forall $v \in (V - W)$
      $d[v] := min\{d[v], w(u, v)\}$
  od
end

Minimální kostra: Primův Algoritmus

Limity paralelizace

- Souběžné zpracování více pivotů je nekorektní
- Iterace while musí být serializovány

Paralelizace

- Mapování vrcholů na procesy
- Jednotlivé procesy identifikují kandidáty na pivota
- ATO redukce kandidátů
- OTA broadcast pivota
- Lokalizovaný update prvků $d[v]$ (cyklus forall)

Nejkratší cesty z jednoho vrcholu: Dijkstrův algoritmus

proc *DIJKSTRA_SINGLE_SOURCE*(V, E, w, s)
  $W := \{s\}$
  forall $v \in (V - W)$
    if $(s, v) \in E$
      then $l[v] := w(s, v)$
      else $l[v] := \infty$
    fi
  while $(W \neq V)$ do
    vyber u tak, aby $l[u] = min\{l[v] \mid v \in V - W\}$
    $W := W \cup \{u\}$
    forall $v \in (V - W)$
      $l[v] := min\{l[v], l[u] + w(u, v)\}$
  od
end

Nejkratší cesty mezi všemi vrcholy: Dijkstrův algoritmus

Princip

- Násobné provedení Dijkstrova algoritmu pro výpočet nejkratších cest vedoucích z jednoho vrcholu

Paralelizace dělením vstupu

- Vrcholy grafu mapovány na procesy
- Každý proces provádí sekvenční výpočet nejkratších cest z vrcholů jemu přidělených
- Při replikaci dat, není třeba komunikace
- Neškáluje pro $p > n$

Paralelizace dělením vstupu a paralelizací operací

- Vrcholy grafu mapovány na skupiny procesorů
- Každá skupina procesorů provádí paralelní algoritmus pro detekci nejkratších cest z jednoho vrcholu
- Škáluje i pro $p > n$

Nejkratší cesty mezi všemi vrcholy: Floydův algoritmus

proc *FLOYD_ALL_PAIRS*(Matice sousednosti A)
  $D^0 := A$
  for $k := 1$ to n do
    for $i := 1$ to n do
      for $j := 1$ to n do
        $d_{i,j}^{(k)} := min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$
      od
    od
  od
end

Nejkratší cesty mezi všemi vrcholy: Floydův algoritmus

Paralelizace

- Pro výpočet prvků $d^{(k)}$ je třeba prvky $d^{(k-1)}$
- Vzhledem k datovým závislostem nelze paralelizovat vnější for cyklus
- Dekompozice vnitřních cyklů dle hodnot proměnných i a j
- Při p procesech se matice velikosti $n \times n$ dekomponuje na $\sqrt{p} \times \sqrt{p}$ bloků o velikosti $n/\sqrt{p} \times n/\sqrt{p}$

Návrh implementace

- Pro výpočet prvku $d_{i,j}$ potřeba prvky z řádku j a sloupku i
- Procesy odešlou jimi zpravovaná data verze $k-1$ ostatním procesům ve stejném řádku či sloupku, poté přijmou data verze $k-1$ od těchto procesů, a spočítají hodnoty verze k

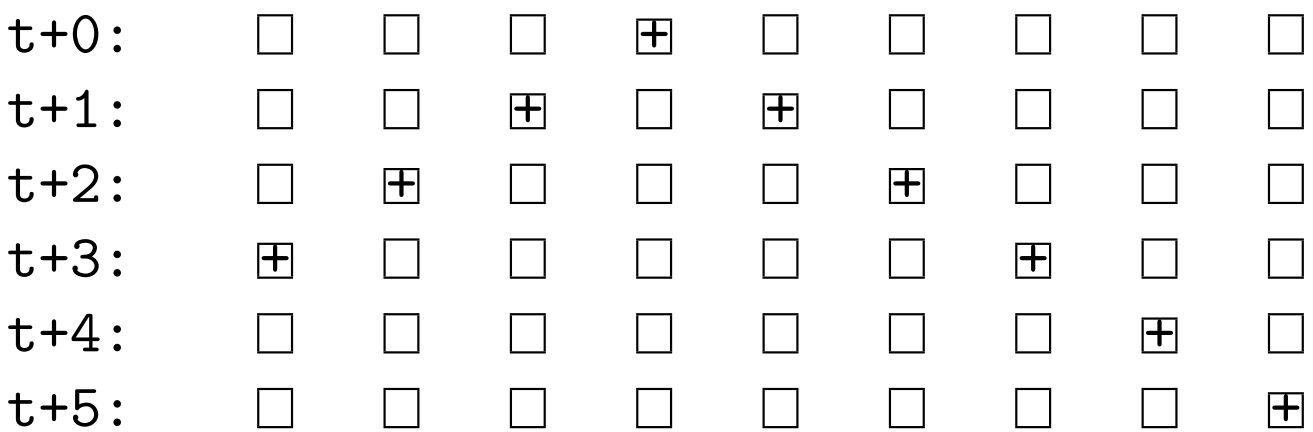
Nejkratší cesty mezi všemi vrcholy: Floydův algoritmus

Naivní implementace

- Prostoje při čekání data od všech okolních procesů

Implementace s použitím modelu pracovní linky (pipeline)

- Proces $P_{i,j}$ se při komunikaci nesynchronizuje s ostatními, nečeká až všichni obdrží data z předchozí iterace



Tranzitivní uzávěr (relace dosažitelnosti)

Princip výpočtu

- Matice dosažitelnosti (matice hodnot typu Bool)
- Při ohodnocení hran hodnotou 1, lze použít libovolný algoritmus pro výpočet nejkratších cest mezi všemi vrcholy

Dijkstrův či Floydův algoritmus

- Nenulová hodnota $d_{i,j}$ znamená existenci cesty

Modifikace Floydova algoritmu

- $d_{i,j}^{(k)} := d_{i,j}^{(k-1)} \vee (d_{i,k}^{(k-1)} \wedge d_{k,j}^{(k-1)})$

Řídké grafy

Výhody

- Algoritmy mohou pracovat bez maticové reprezentace
- Složitosti algoritmů přestávají mít dolní odhad V^2
- Složitosti algoritmů mají dolní odhad $V + E$

Paralelizace

- Celkový výkon paralelního algoritmu výrazně ovlivněn dělením grafu

Řídké grafy: Maximální nezávislá množina

Definice

- Vrcholy u, v jsou nezávislé, pokud neexistuje hrana (u, v)
- Maximální nezávislá množina vrcholů, je množina po dvou nezávislých vrcholů, kterou nelze rozšířit o další vrchol tak, aby zůstala nezávislá

proc *MAXIMAL_INDEPENDENT_SET*(V, E)
  $I := \emptyset$; $C := V$
  while $(C \neq \emptyset)$ do
    vyber vrchol u z množiny kandidátů C
    $I := I \cup \{u\}$
    forall $(v \in C)$ if $(u, v) \in E$
      then $C := C - \{v\}$
    fi
  od
end

Řídké grafy: Maximální nezávislá množina

Problém paralelizace

- Jak souběžně zvolit více vrcholů, aby byla garantována jejich nezávislost?

Řešení

1) Vrcholům se přiřadí náhodná čísla
2) Paralelně se identifikují vrcholy, jejichž všichni sousedé v množině kandidátů mají přiřazena čísla větší než je číslo testovaného vrcholu
3) Identifikované vrcholy se přidají do nezávislé množiny a jejich sousedé se odstraní z množiny kandidátů
4) Opakuje se 2)-3) dokud není množina kandidátů prázdná

Řídké grafy: Nejkratší cesty z jednoho vrcholu

Modifikace Dijsktrova algoritmu

- Pro určení vrcholu u s minimální hodnotou $l[u]$ lze použít prioritní frontu, která upřednostňuje vrcholy s menší hodnotou l
- Frontu lze udržovat za cenu $log\ n$ (binární halda)
- Operaci pro aktualizaci hodnot sousedů vybraného vrcholu u lze provést efektivně pomocí seznamu následníků

Paralelizace zachovávající optimální množství práce

- Jeden proces udržuje prioritní frontu (extrahuje minima)
- Ostatní procesy provádí aktualizace hodnot l
- Synchronizace po zpracování každého vrcholu
- Je-li $l[u]$ aktuální minimální hodnota vybraného vrcholu, je korektní souběžně zpracovávat všechny vrcholy v takové, že $l[v] = l[u]$, nebo přesněji $l[v] \leq l[u] + m$, kde m je minimální cena hrany

Řídké grafy: Nejkratší cesty z jednoho vrcholu

Paralelizace nezachovávající optimální množství práce

- Souběžné zpracování všech vrcholů z fronty
- Některým vrcholům v může být přiřazena nesprávná hodnota $l[v]$
- Nesprávná hodnota je detekována v okamžiku výpočtu správné hodnoty
- Vrcholy v, jejichž hodnota je nesprávná, jsou znovu zařazeny do fronty se správnou hodnotou $l[v]$
- Některé vrcholy zpracovány opakovaně (dochází k postupnému vylepšování hodnoty $l[v]$)

Řídké grafy: Detekce cyklu v orientovaném grafu

Sekvenční algoritmus

- Prohledávání grafu do hloubky (DFS)
- Přítomnost cyklu detekována hranou vedoucí z aktuálně zkoumaného vrcholu do vrcholu na zásobníku

Paralelní algoritmus

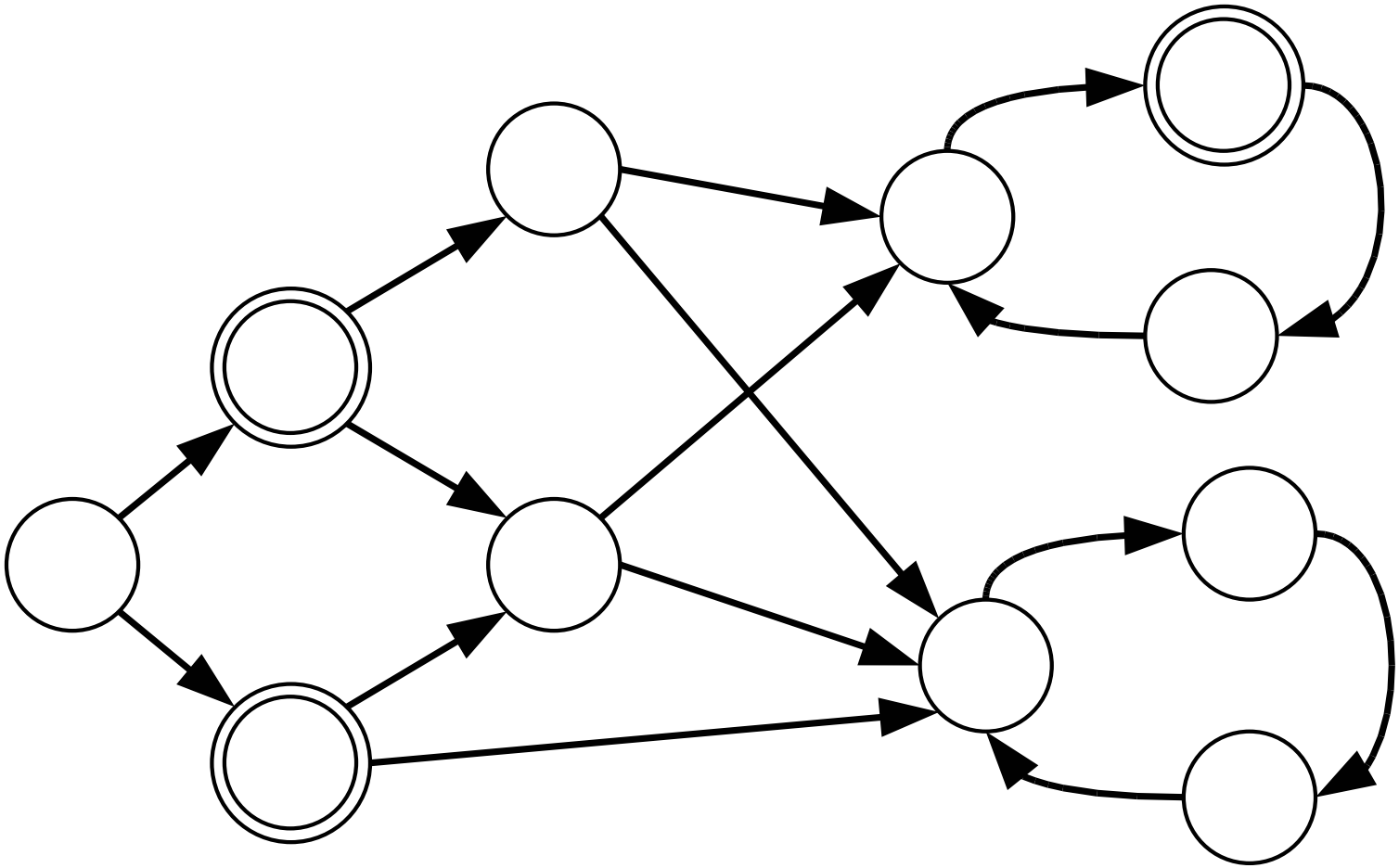
- Lze řešit asymptoticky optimálně bez použití DFS
- Topologické uspořádání: $u < v$ jestliže $(u, v) \in E$
- Vrcholy grafu lze topologicky uspořádat pouze pokud graf neobsahuje cyklus

Řídké grafy: Detekce cyklu v orientovaném grafu

proc $CYCLE\_DETECTION(V, E)$
  $Q := \emptyset$
  $CV := V$
  forall $v \in V$
    $indegree(v) = \|\{(u, v) \in E\}\|$
    if $(indegree(v) = 0)$ then $Q := Q \cup \{v\}$ fi
  while $(Q \neq \emptyset)$ do
    $u := Extract(Q)$
    $CV := CV - \{u\}$
    forall $v \in ADJ(u)$
      $indegree(v) := indegree(v) - 1$
      if $(indegree(v) = 0)$
        then $Q := Q \cup \{v\}$
      fi
  od
  if $(CV \neq \emptyset)$ then $Output($"`Cycle detected`"$)$ fi
end

Detekce akceptujícího cyklu

Akceptující cyklus



Detekce akceptujícího cyklu

Algoritmus *Nested DFS*

- Dvojí prokládané prohledávání do hloubky
  - 1. hledání identifikuje akceptující stavy
  - 2. hledání testuje akceptující stavy na sebe-dosažitelnost
- Pokud se obě DFS procedury správně prokládají, algoritmus je korektní a pracuje v lineárním (optimálním) čase
- 2. hledání je spuštěno v okamžiku opuštění akceptujícího vrcholu 1. procedurou (DFS postorder)
- 2. hledání je omezeno na vrcholy dosažitelné z daného akceptujícího vrcholu a dosud nenavštívené v předchozích voláních 2. procedury

Detekce akceptujícího cyklu

Problém

- Výpočet DFS postorderu je inherentně sekvenční problém
- Optimální paralelní a škálovatelný algoritmus pro výpočet DFS-postorderu není znám (a pravděpodobně neexistuje)
- Nested DFS není použitelné na paralelních platformách

Algoritmus OWCTY

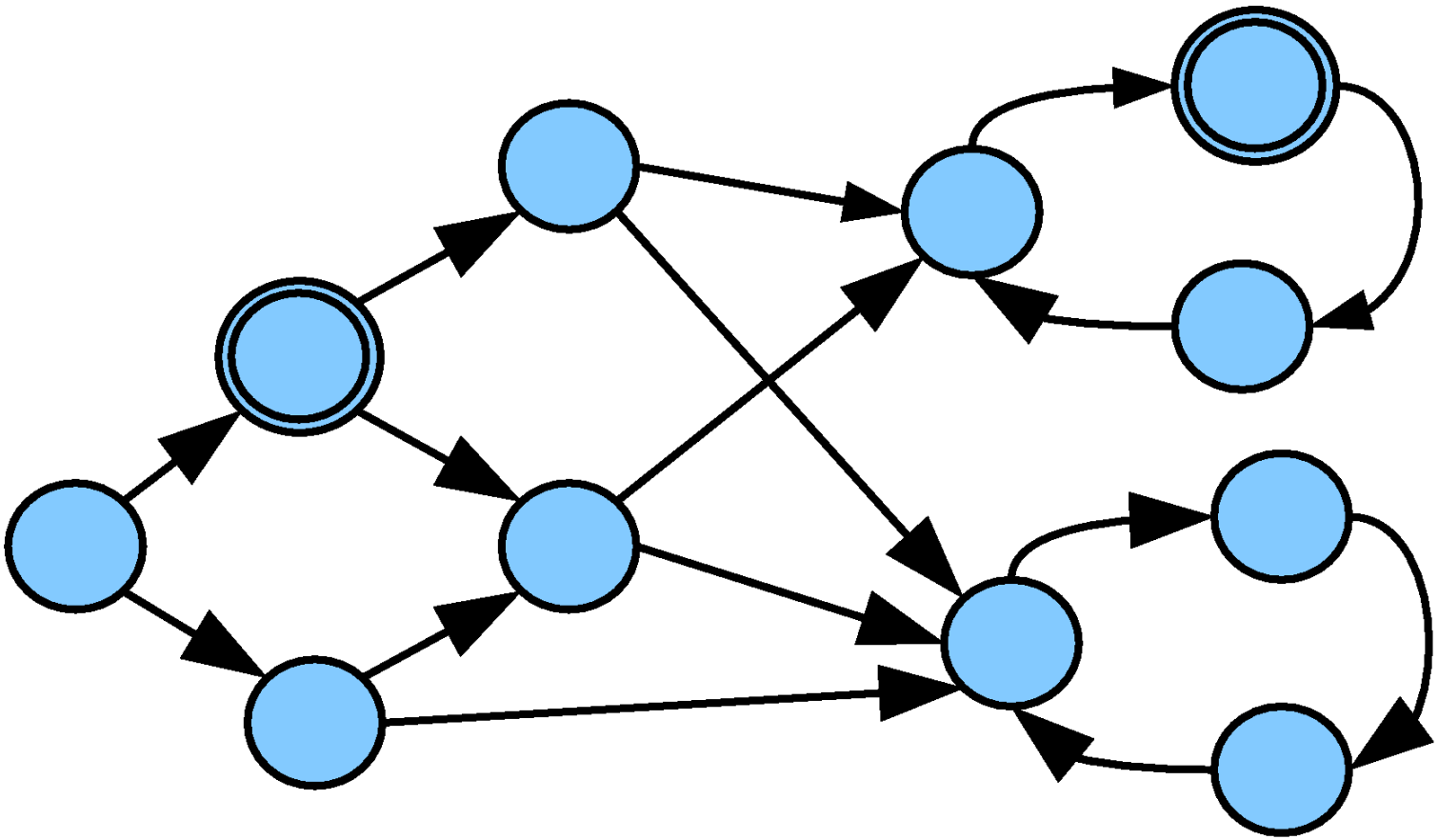
Idea

- Odstranění vrcholů, které nemohou být na akceptujícím cyklu
- Vlastnosti vrcholů na akceptujícím cyklu
  - musí být dosažitelné z nějakého akceptujícího vrcholu
  - musí mít alespoň jednoho předchůdce

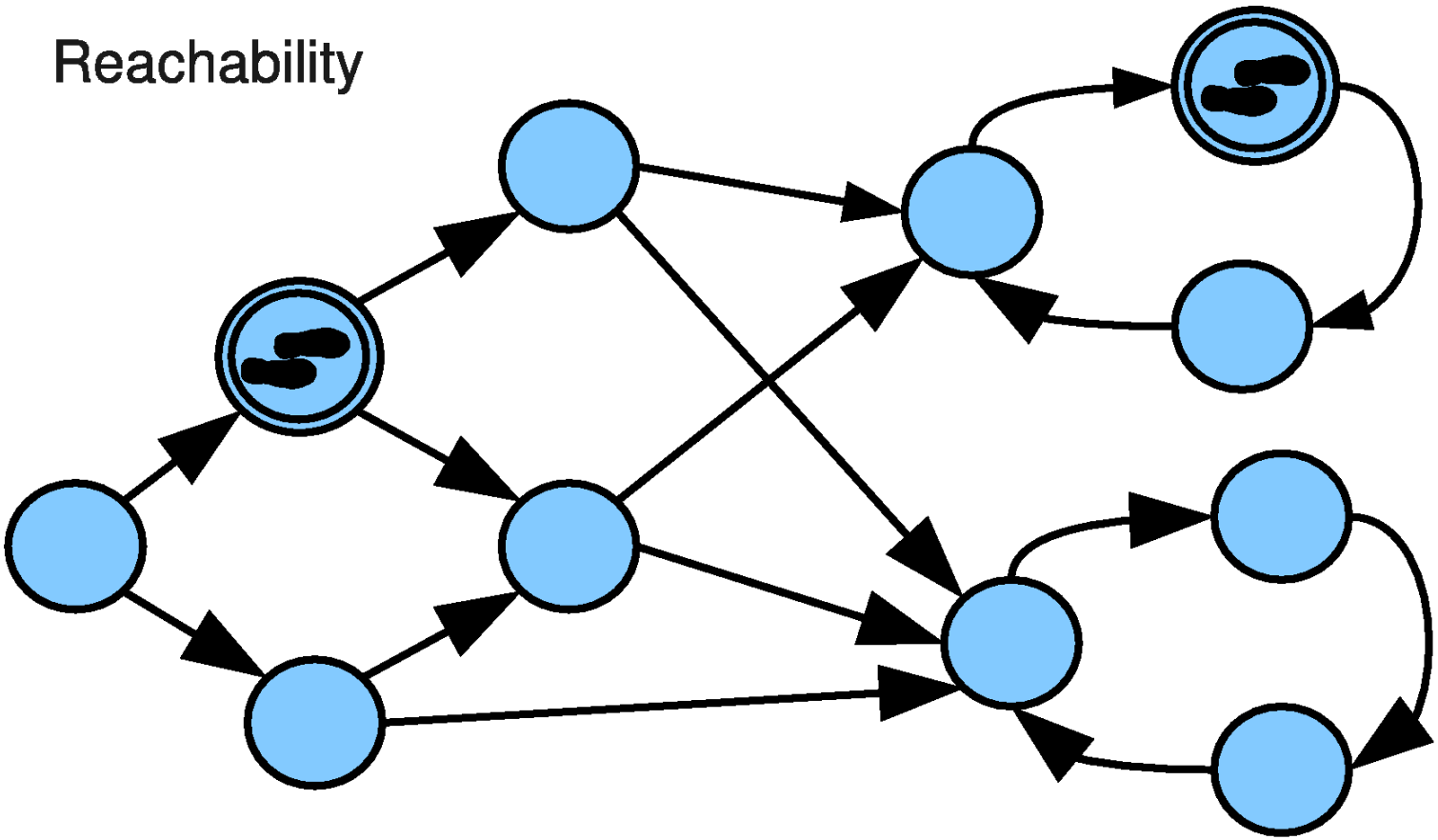
Realizace

- Paralelní procedury pro odstraňování vrcholů
  - Reachability
  - Elimination
- Procedury se opakují dokud se nedosáhne pevného bodu
- Neprázdný graf indikuje přítomnost akceptujícího cyklu

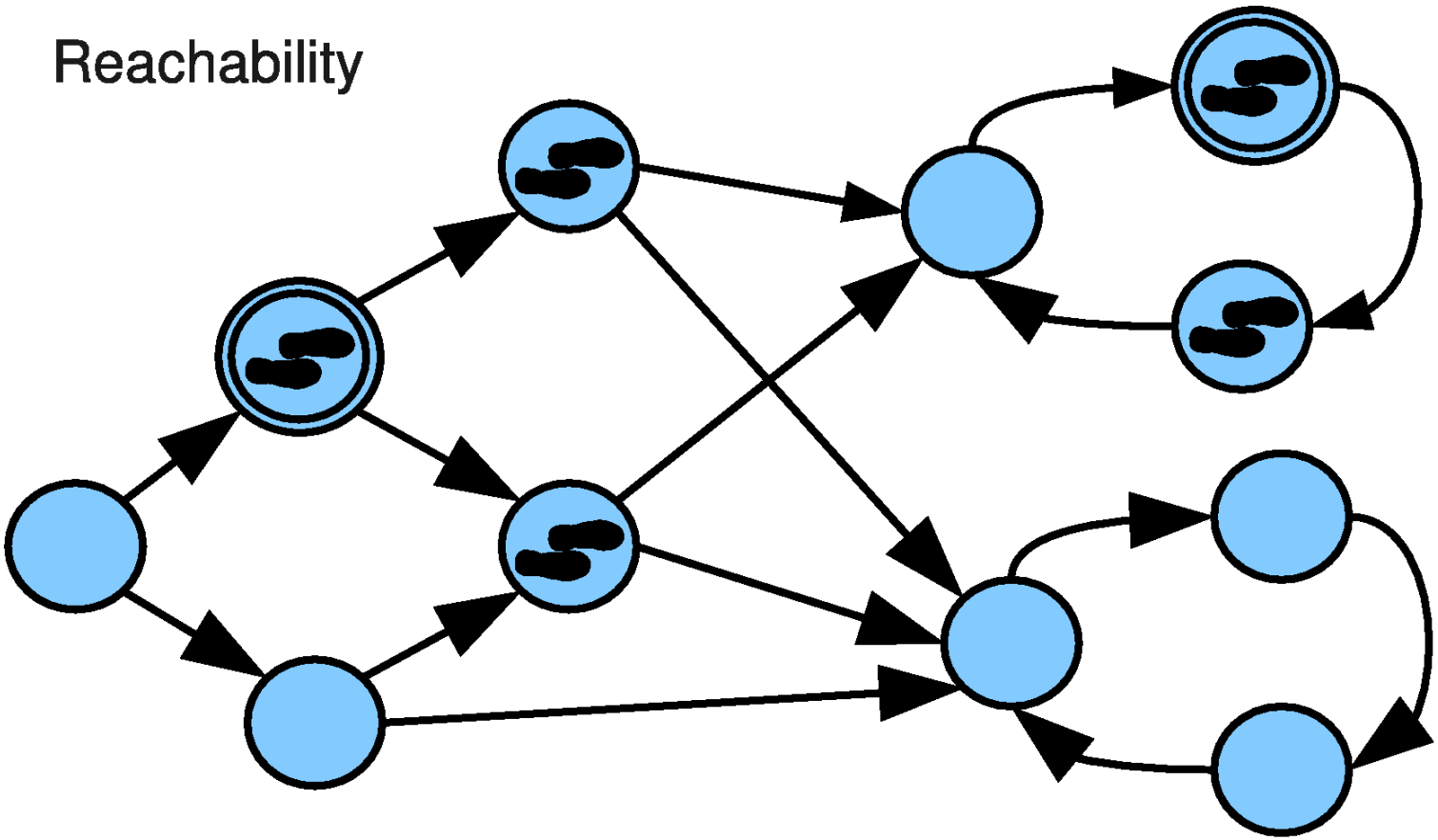
OWCTY



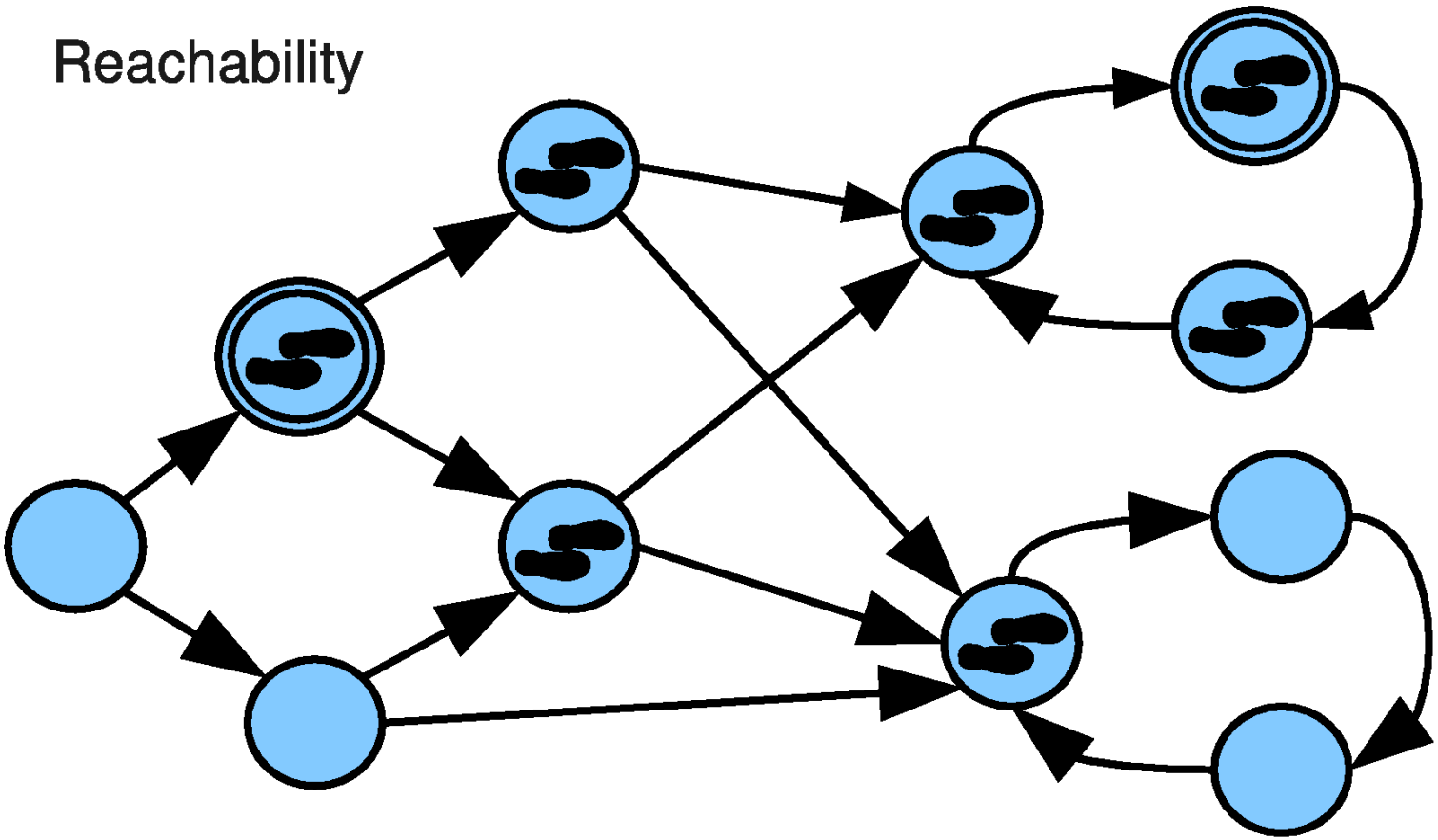
1st iteration



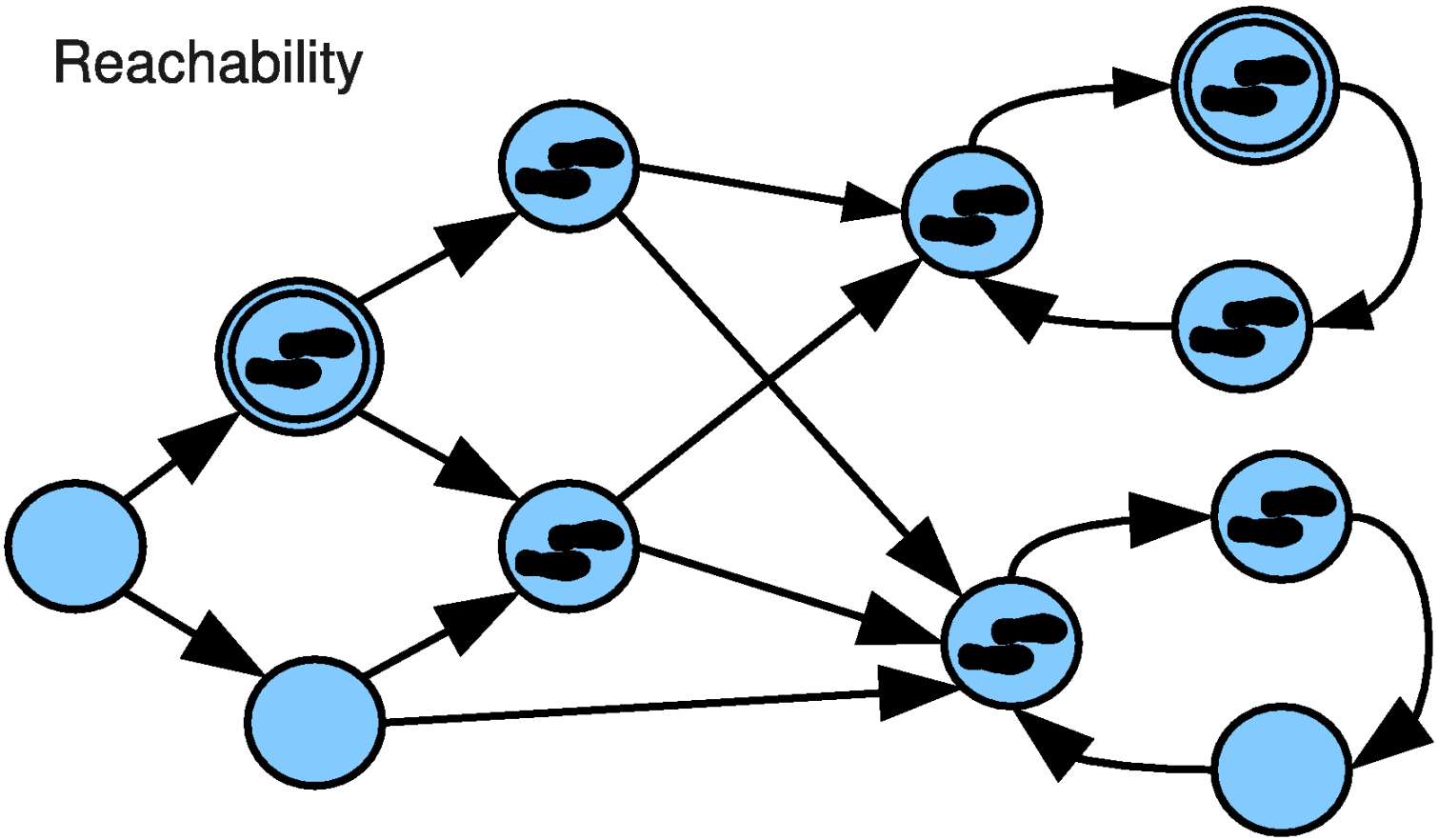
1st iteration



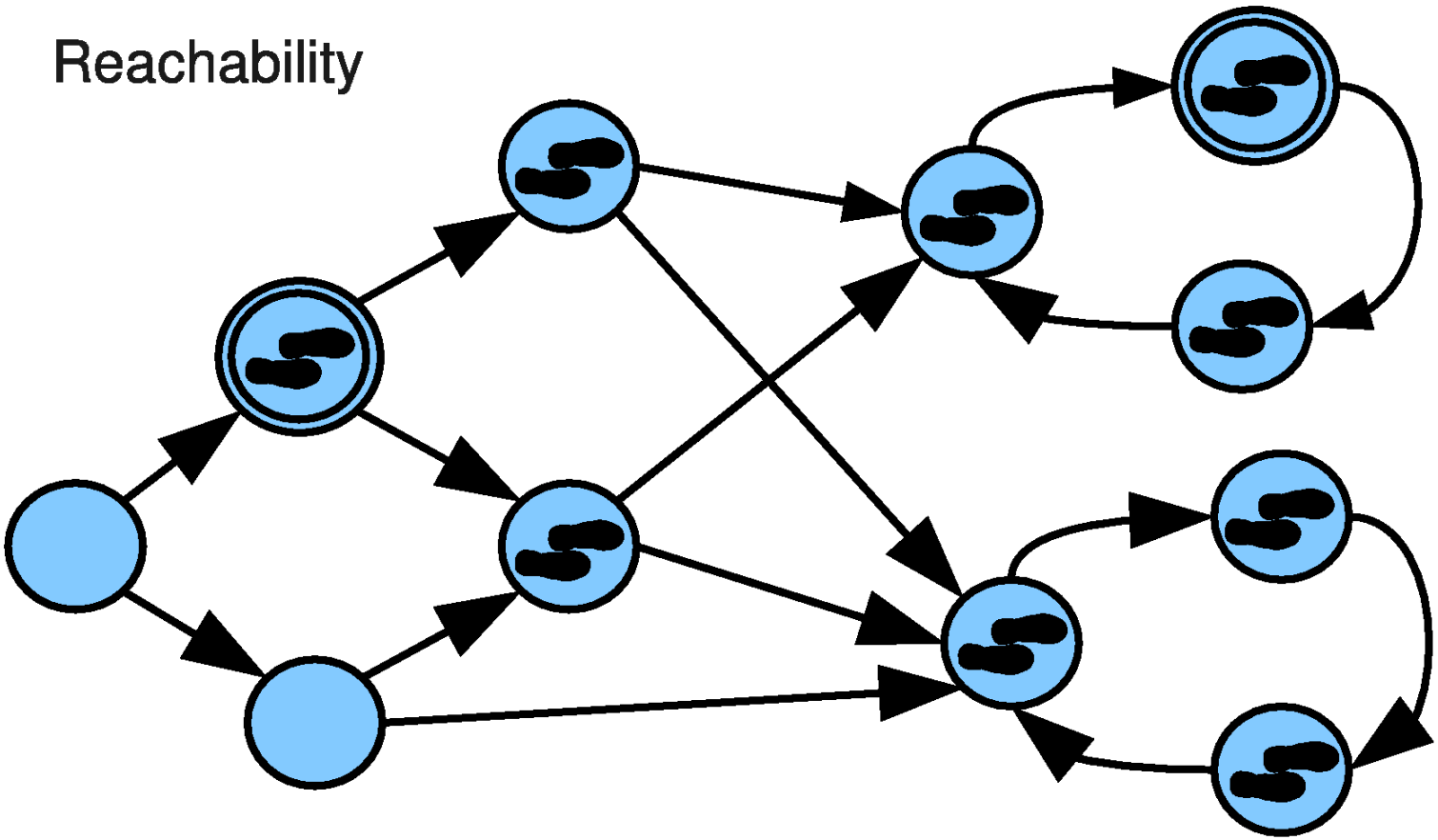
1st iteration



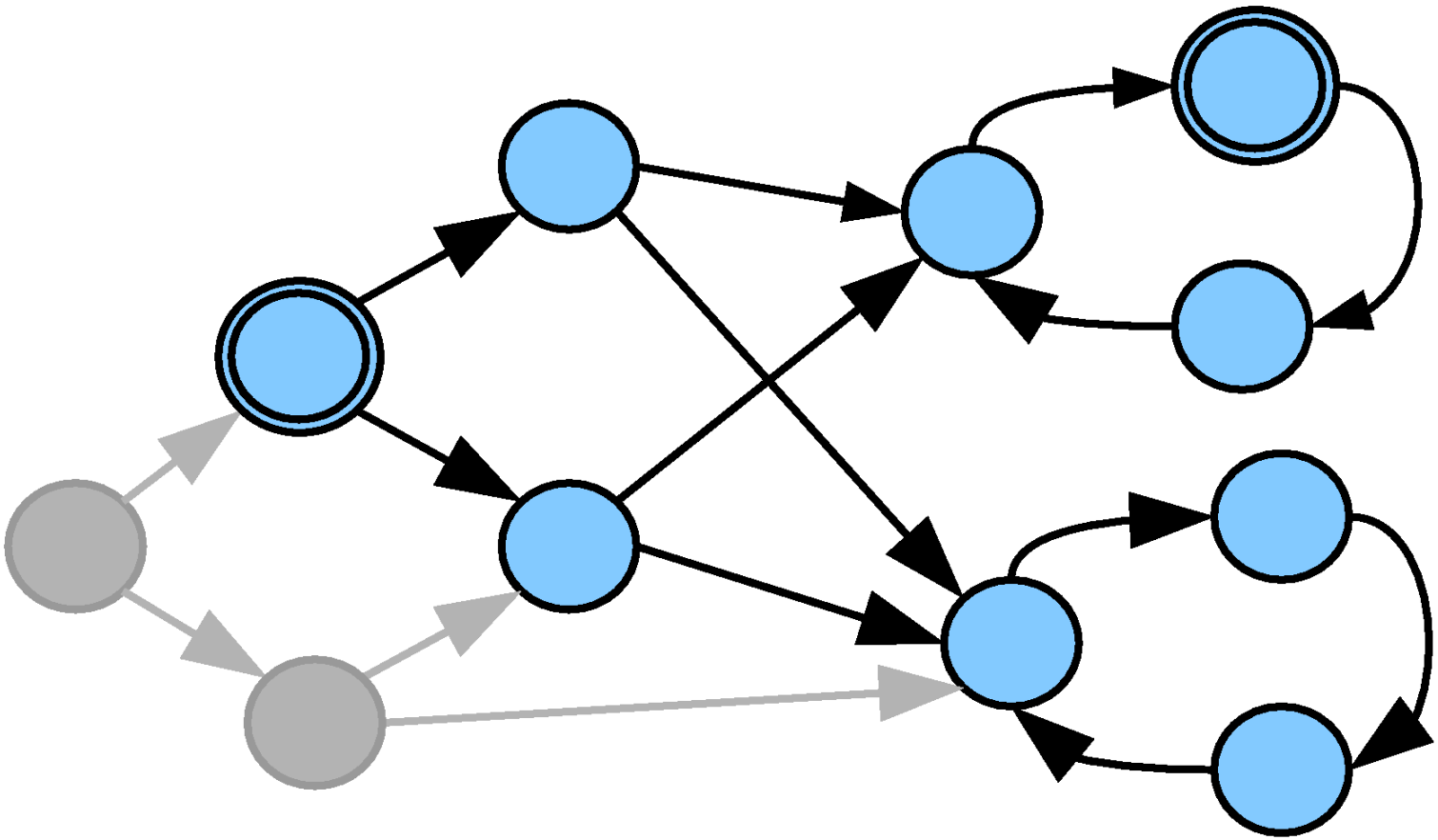
1st iteration



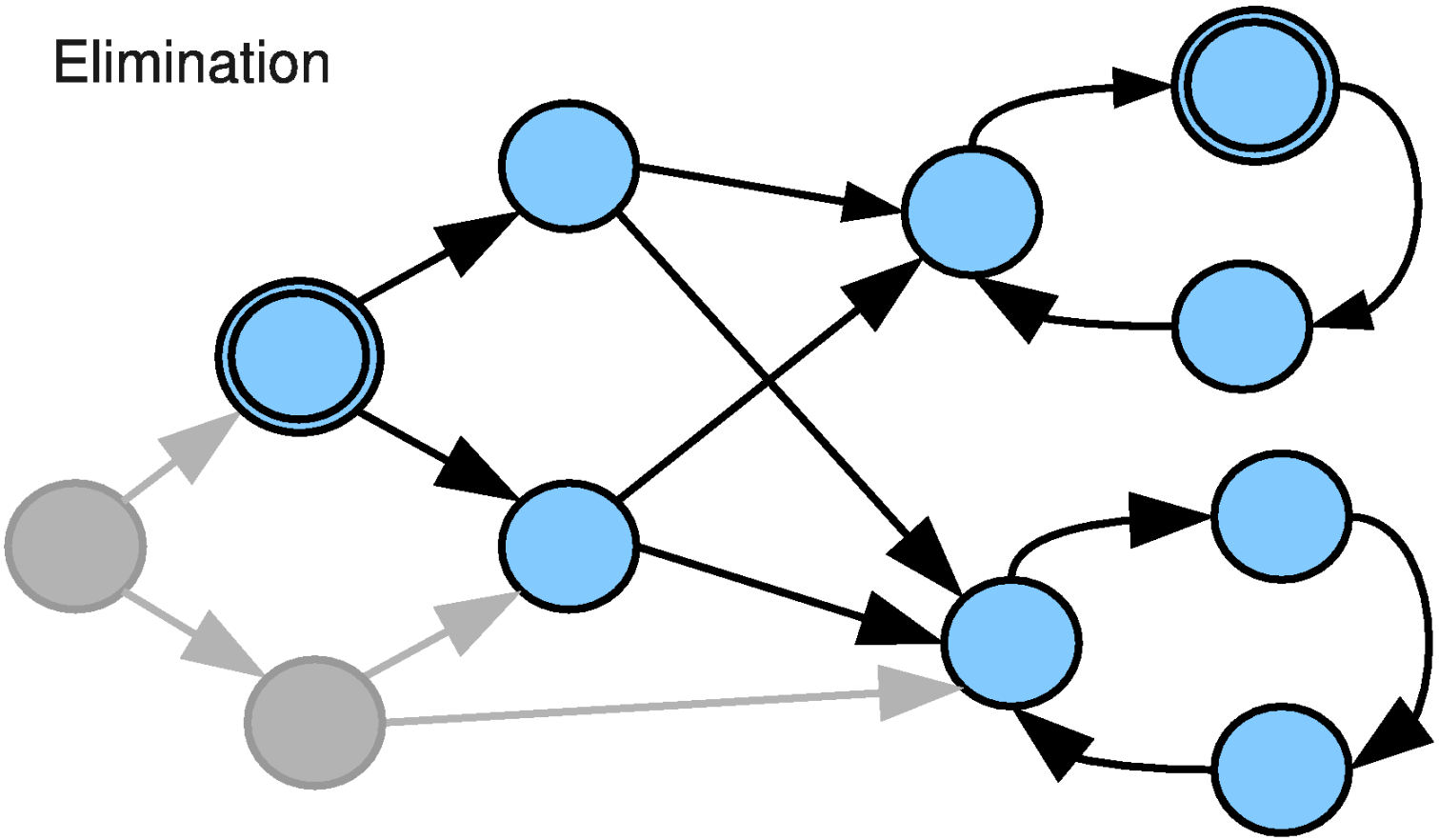
1st iteration



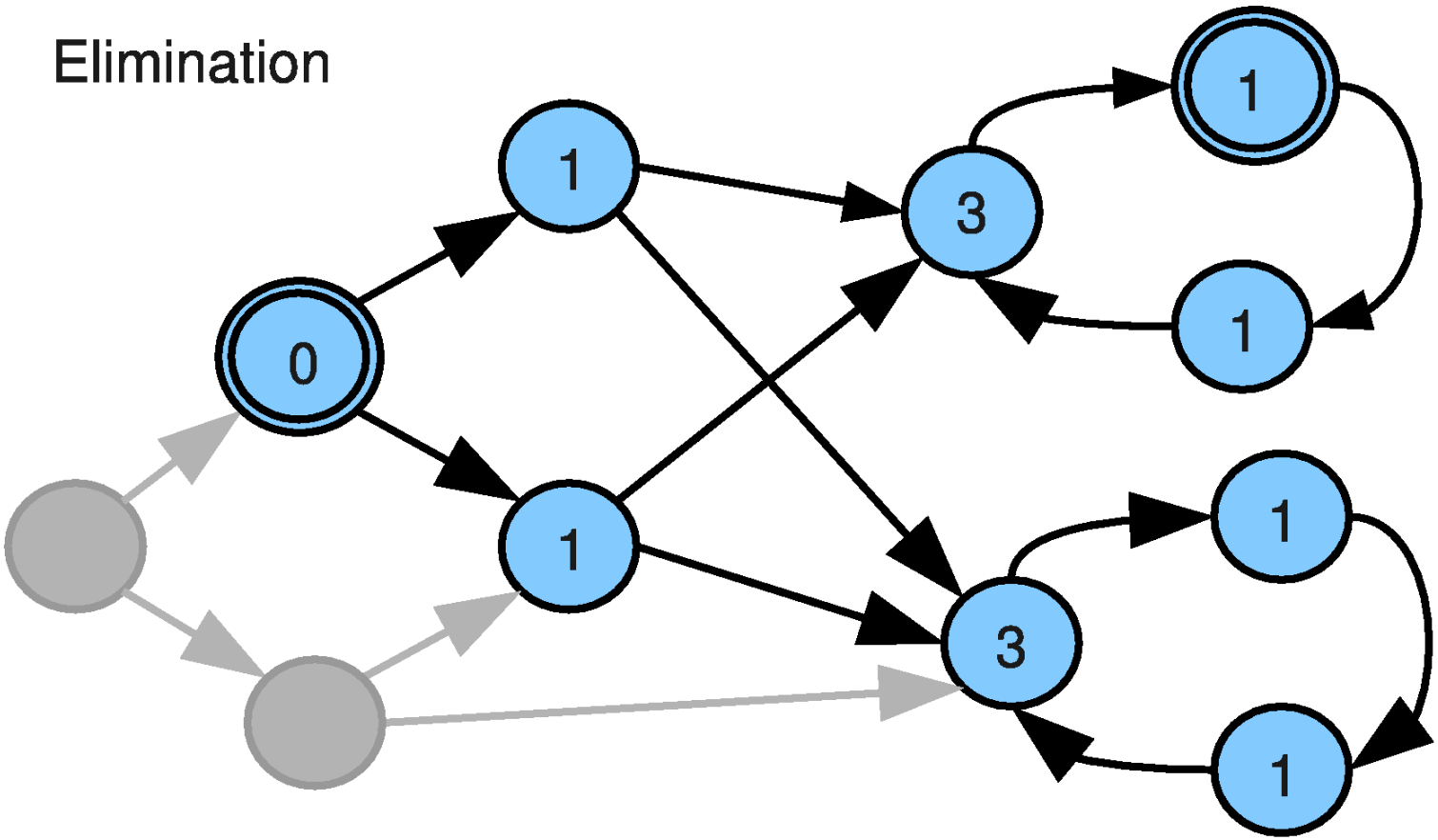
1st iteration



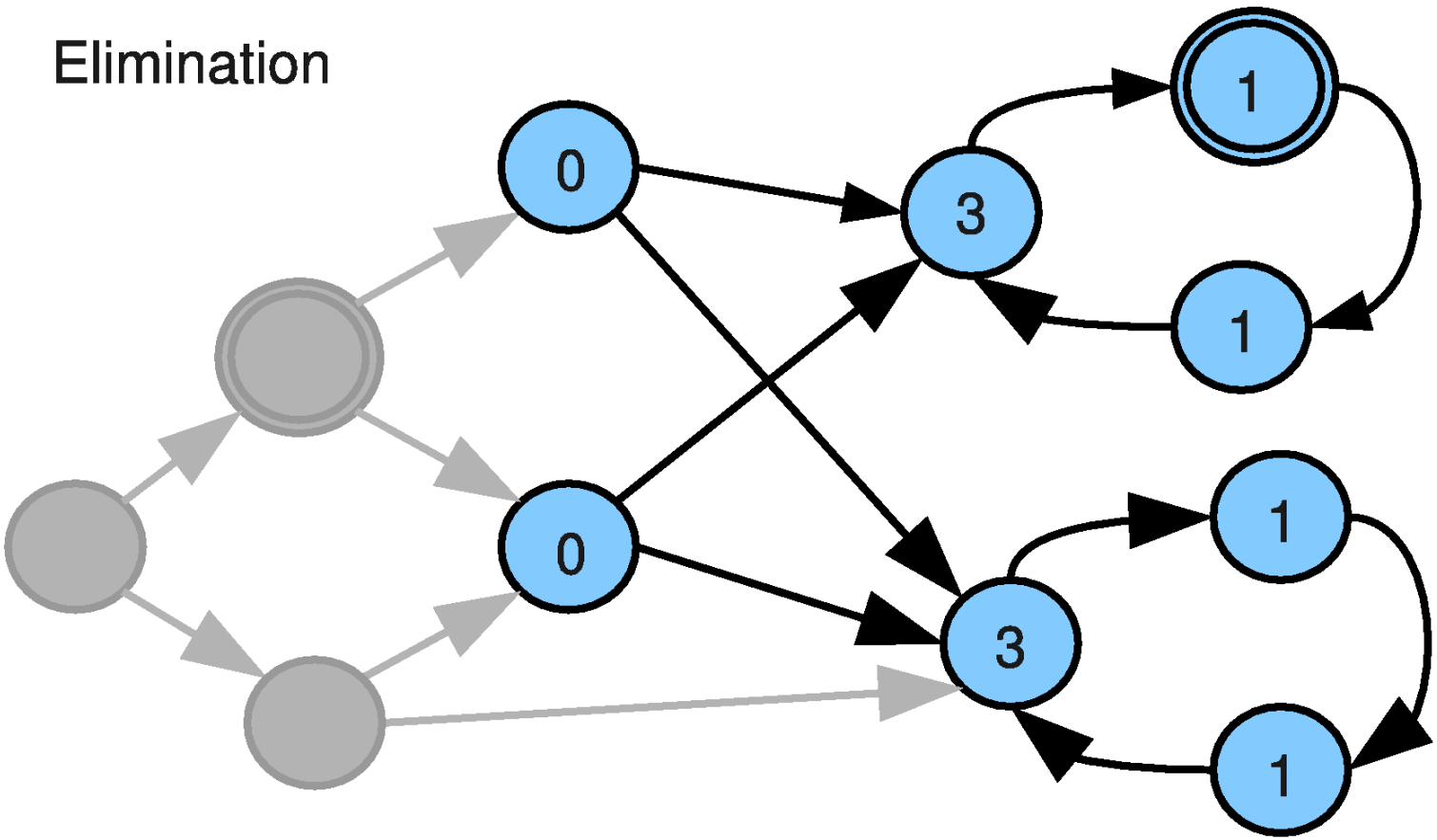
1st iteration



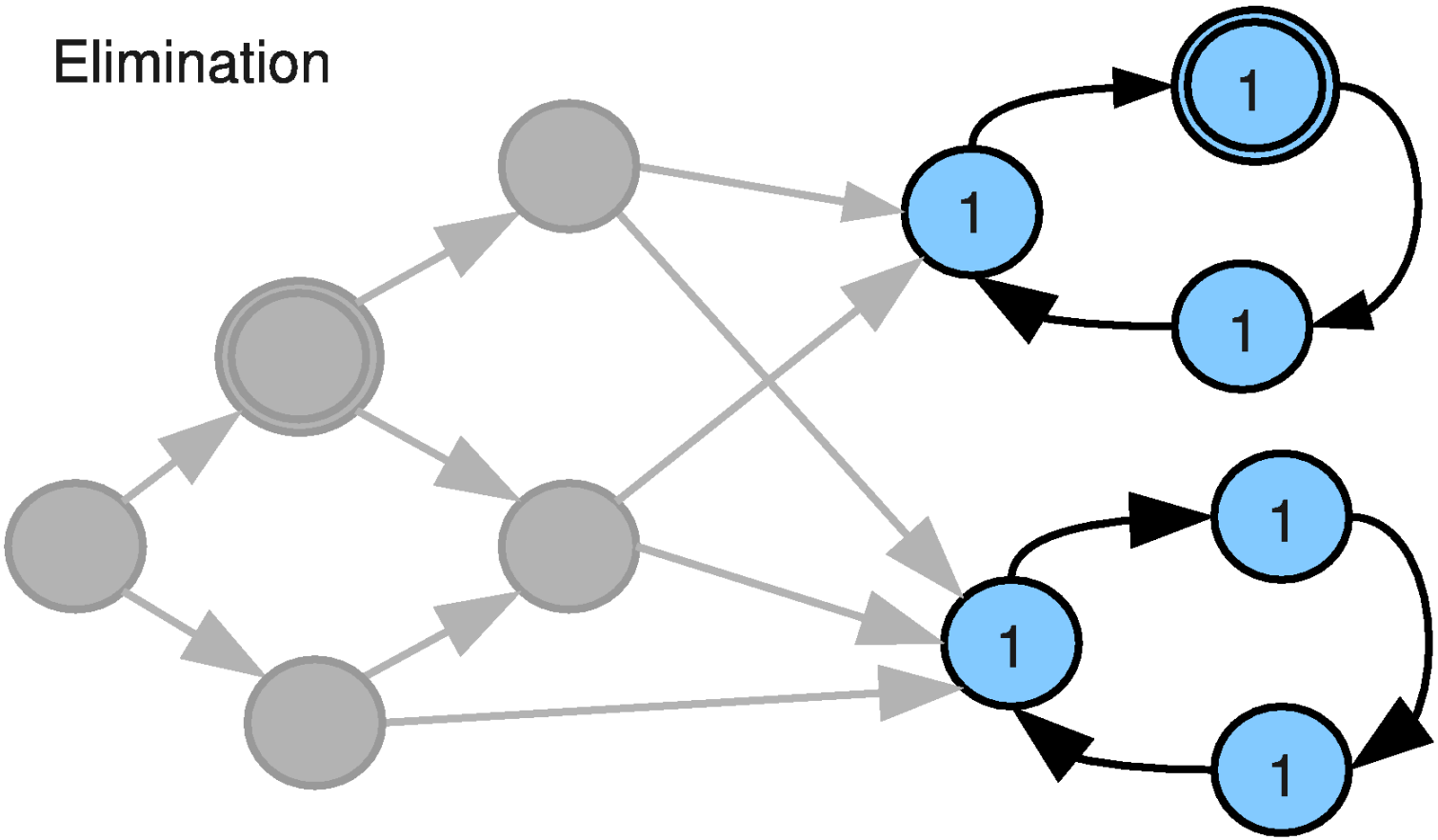
1st iteration



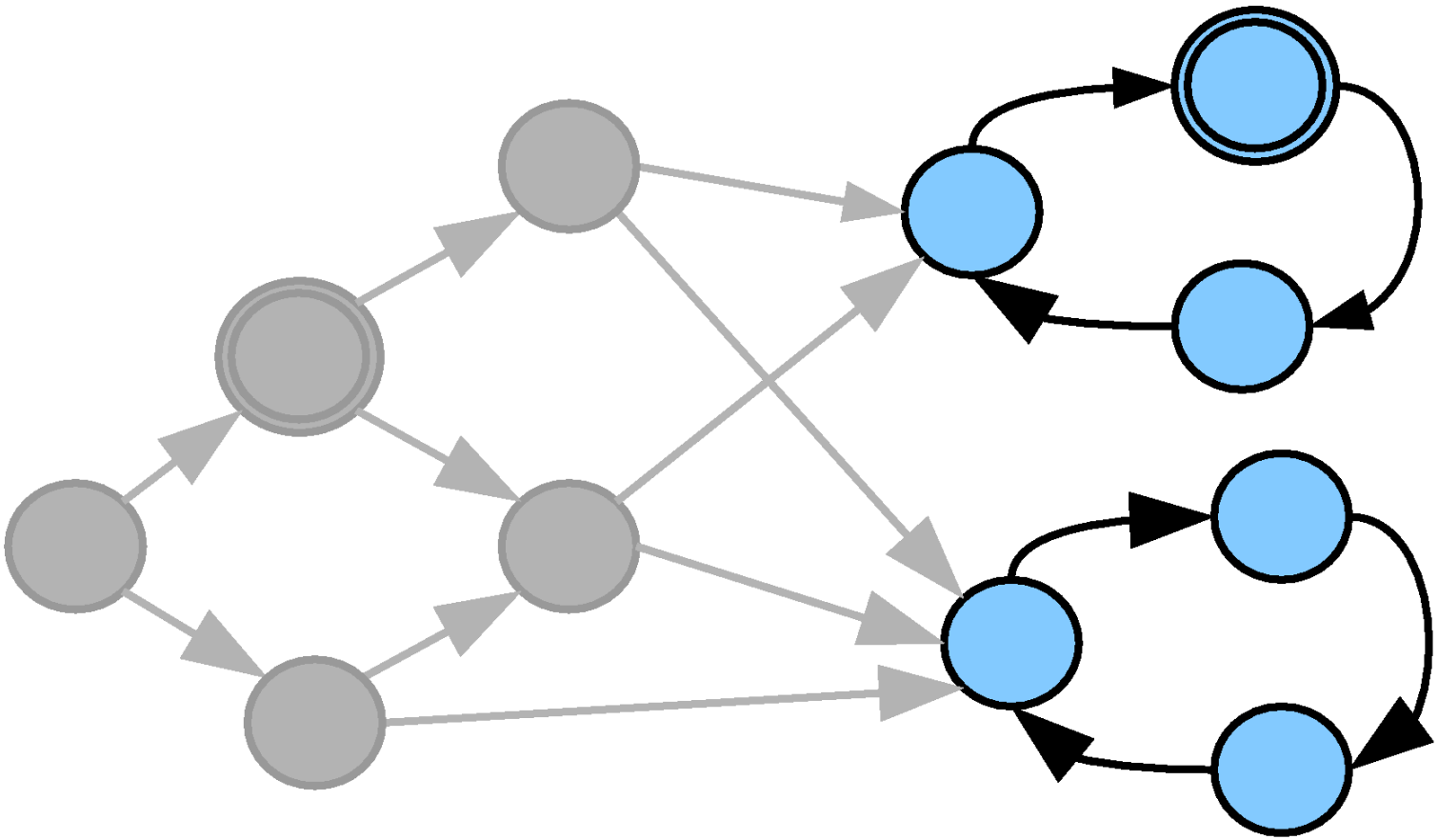
1st iteration



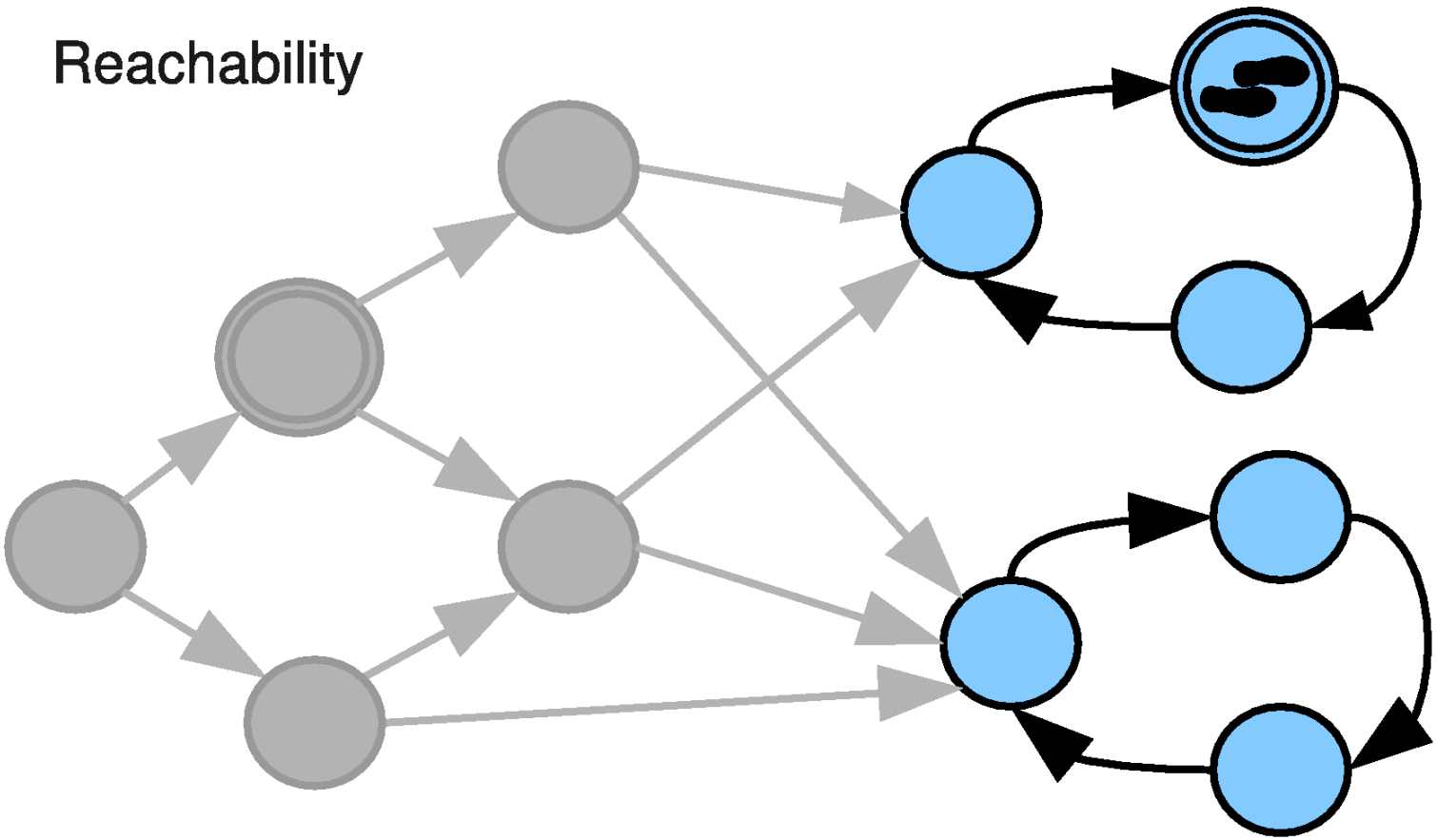
1st iteration



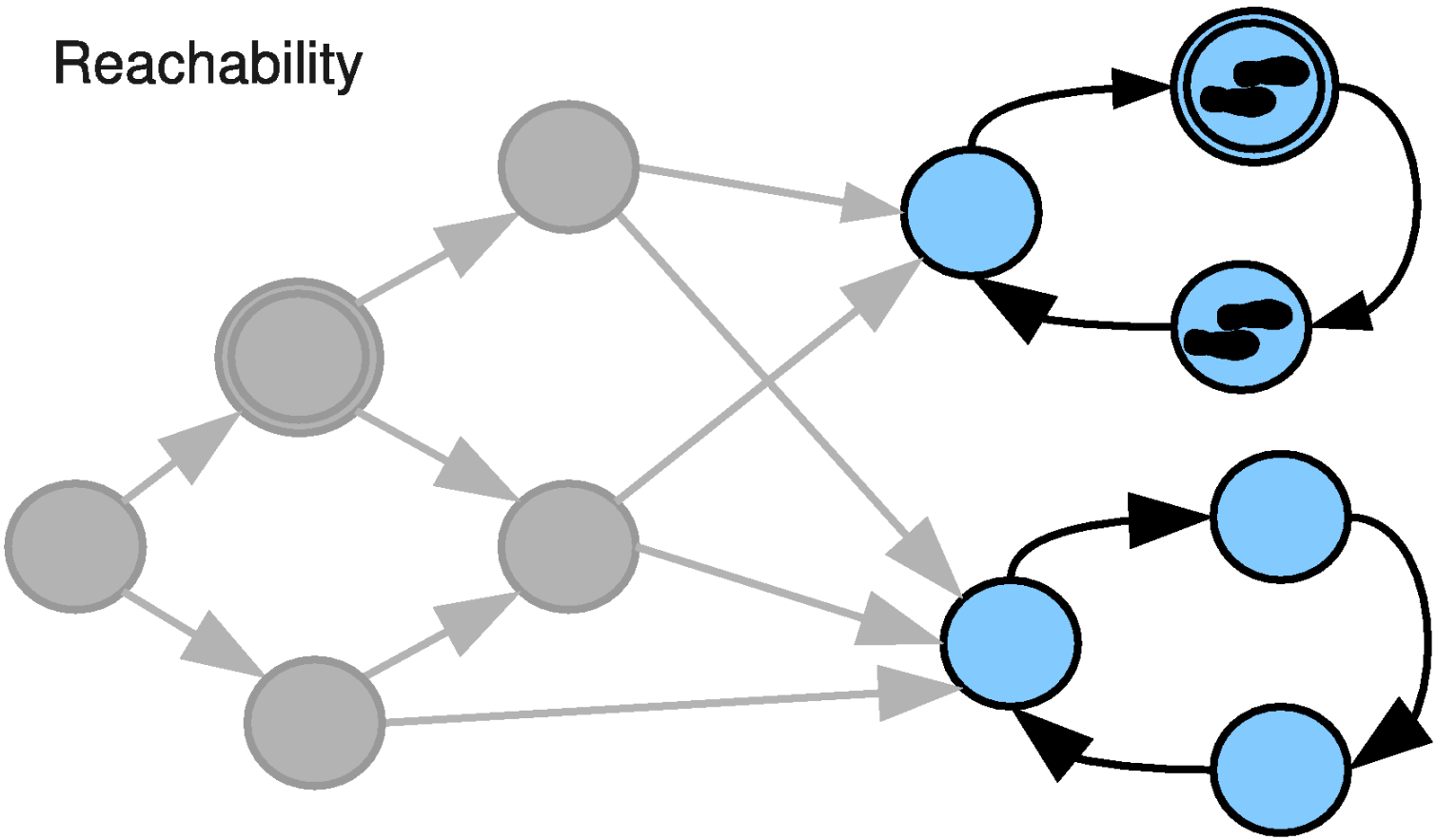
1st iteration



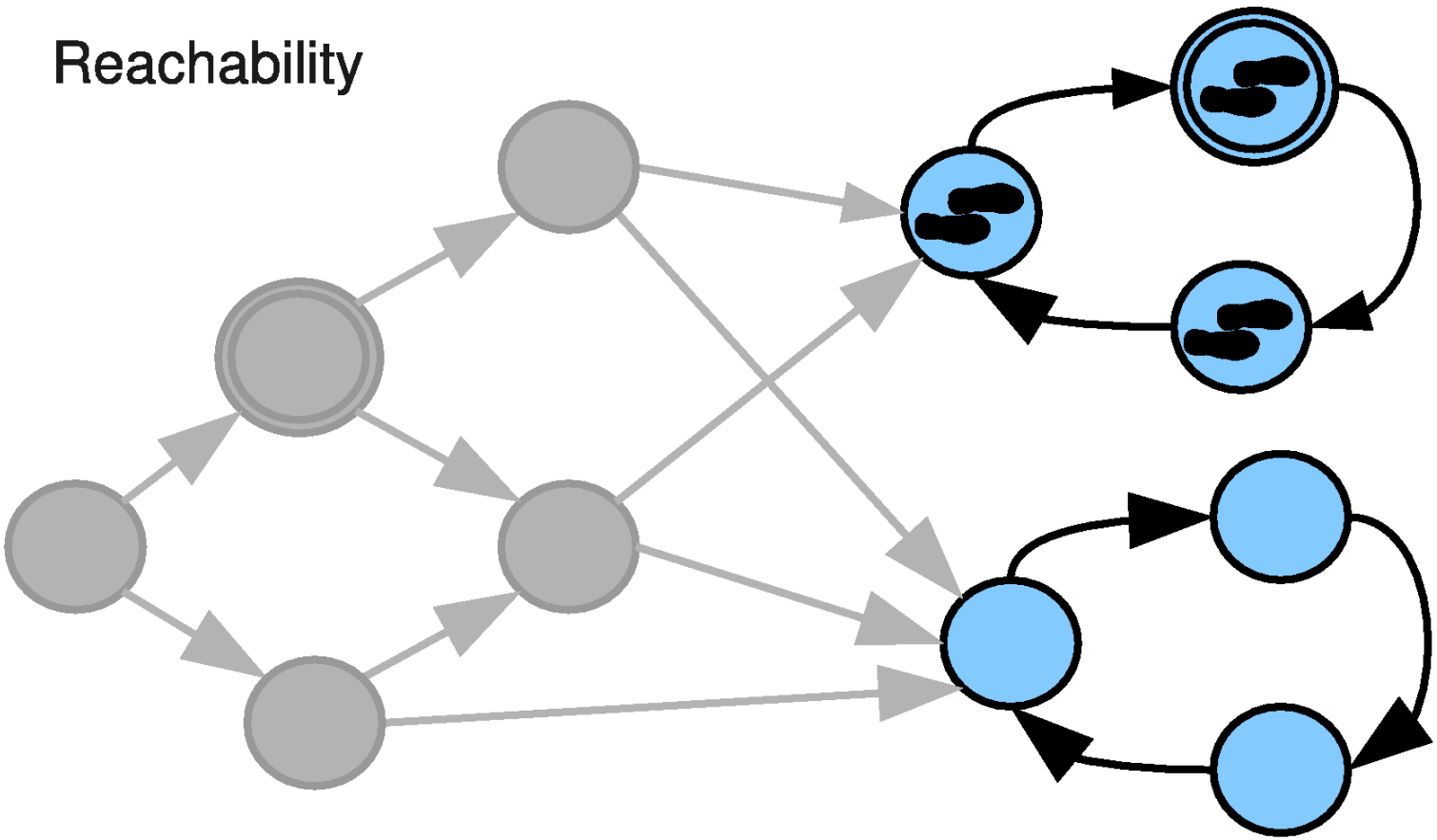
2nd iteration



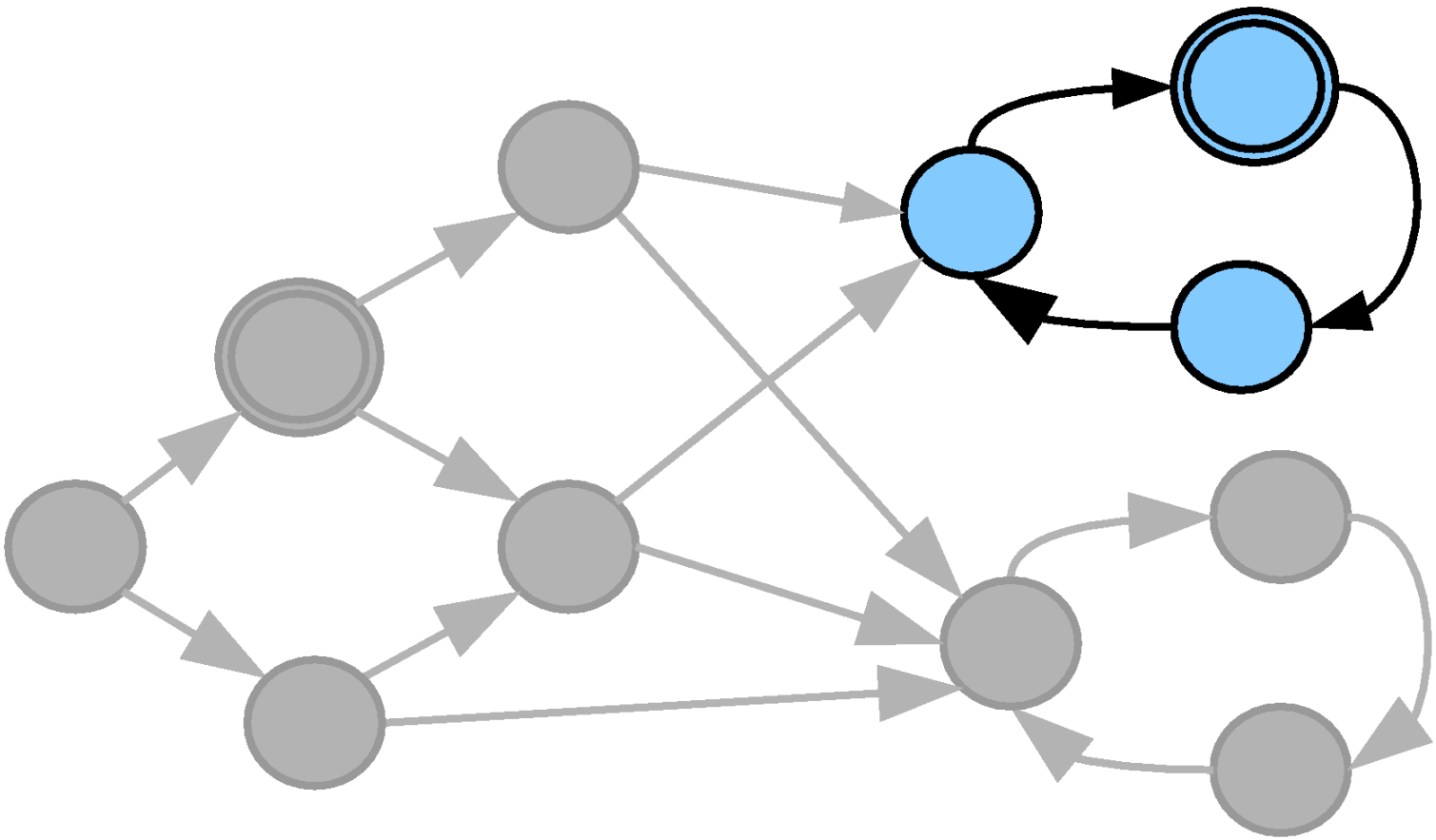
2nd iteration



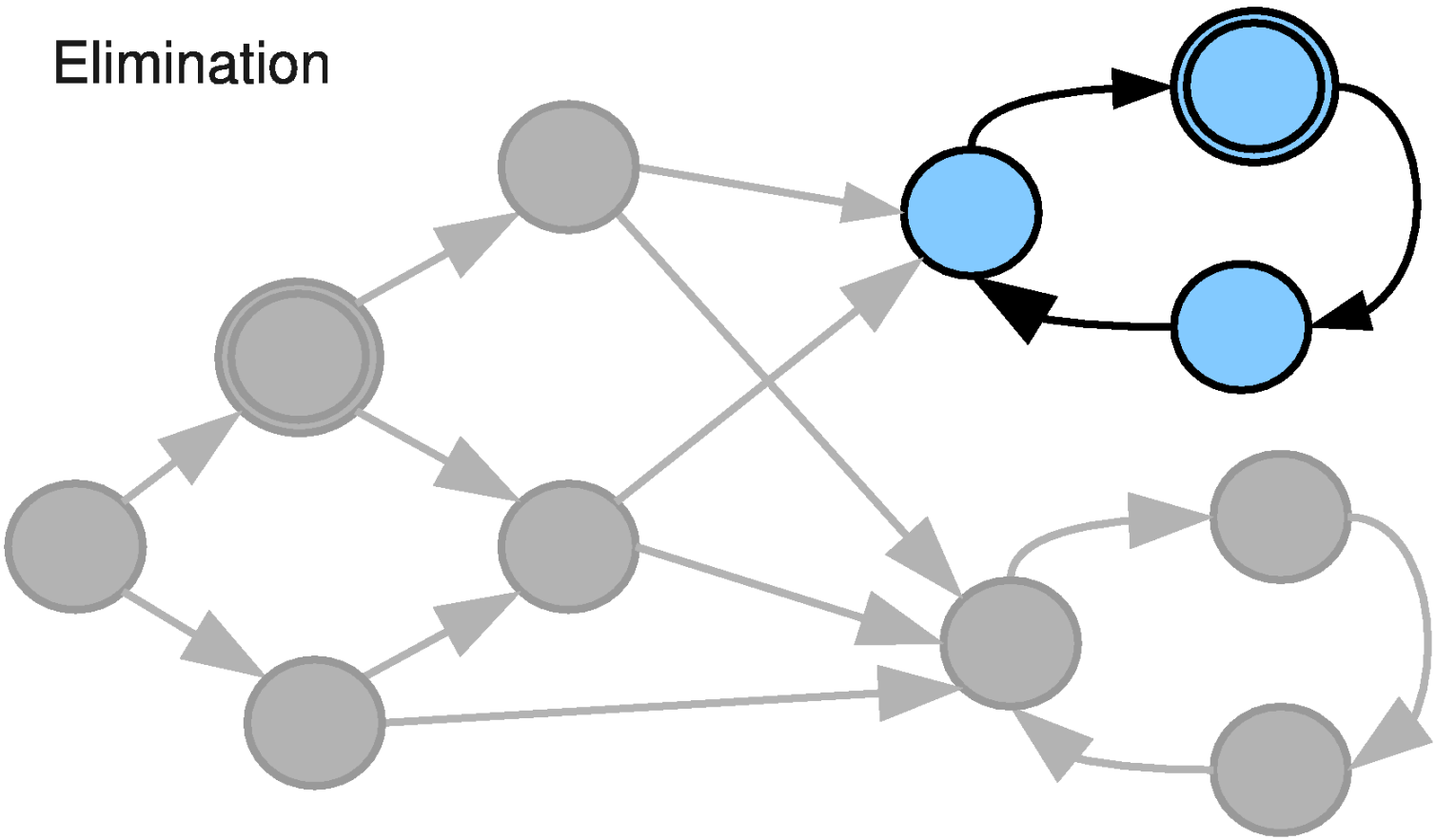
2nd iteration



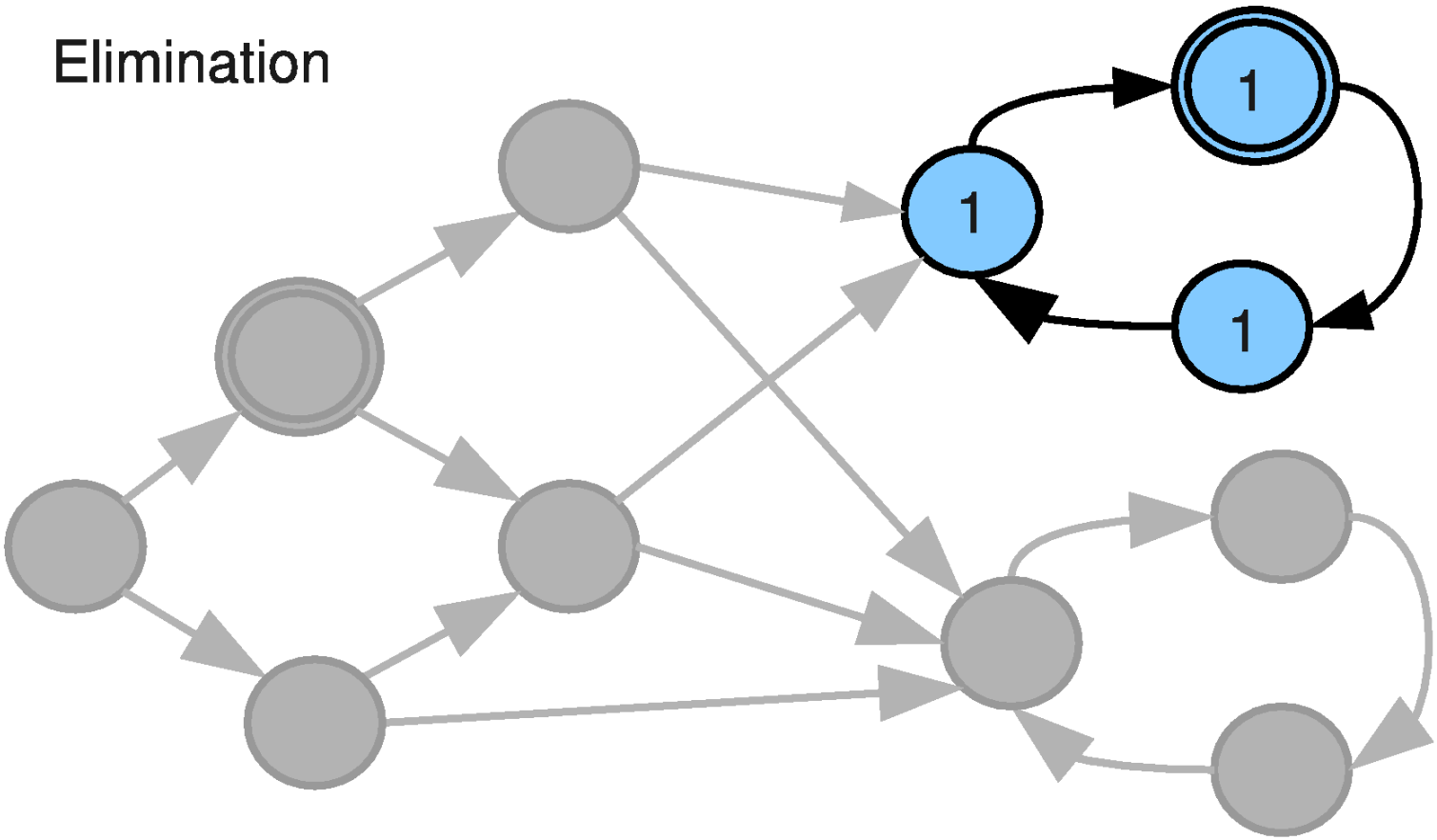
2nd iteration



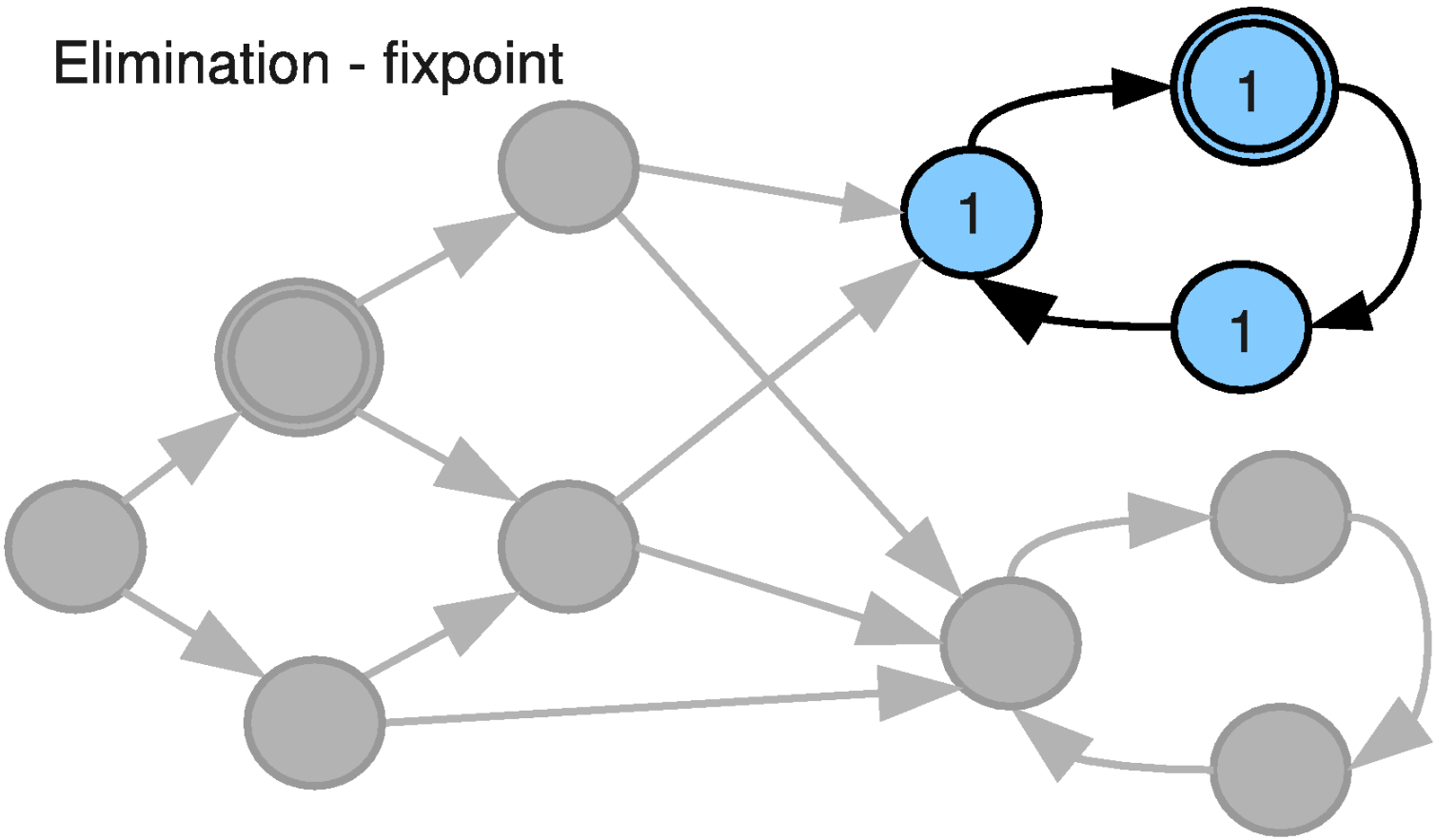
2nd iteration



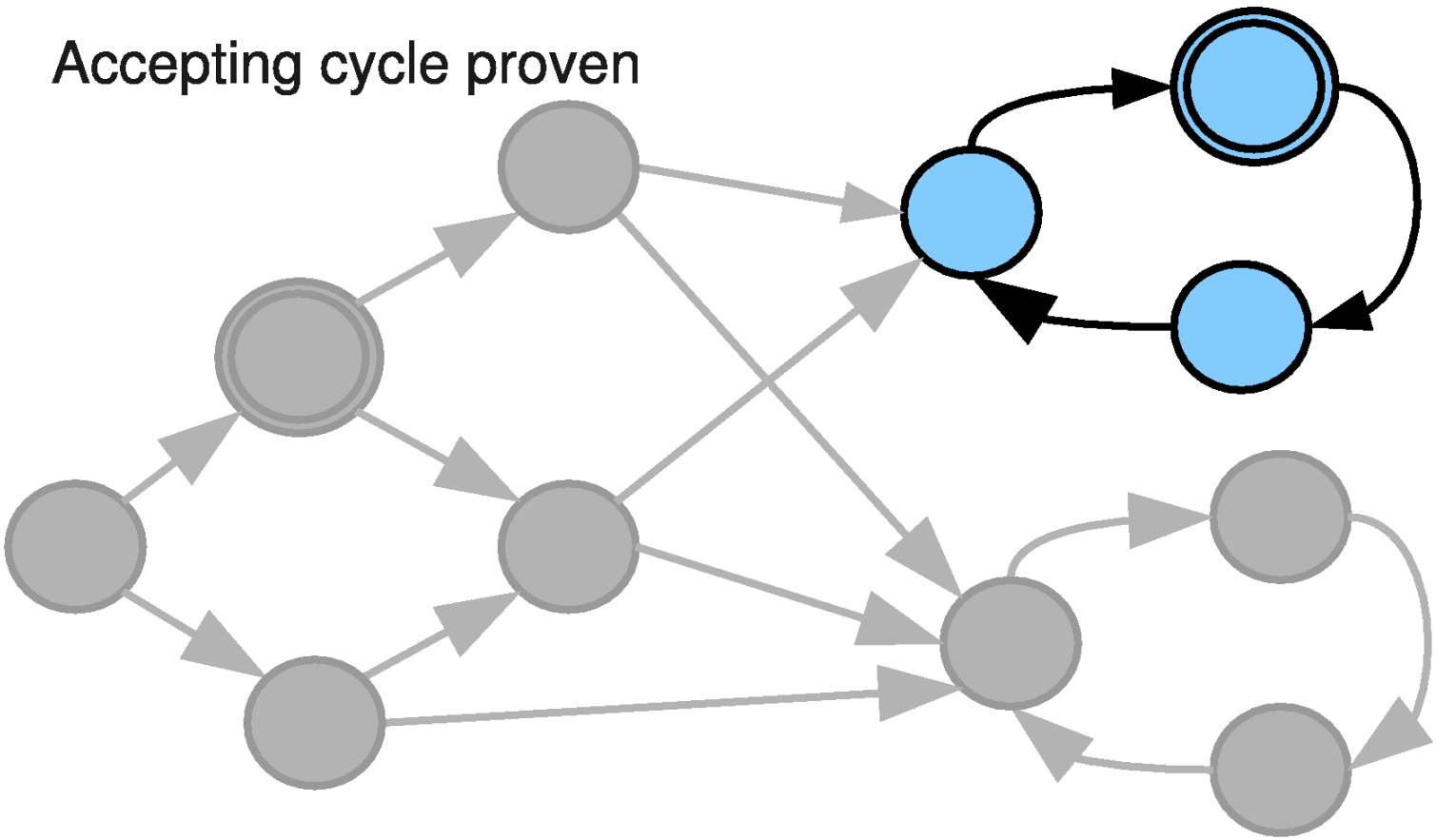
2nd iteration



2nd iteration



OWCTY



MAP Algoritmus

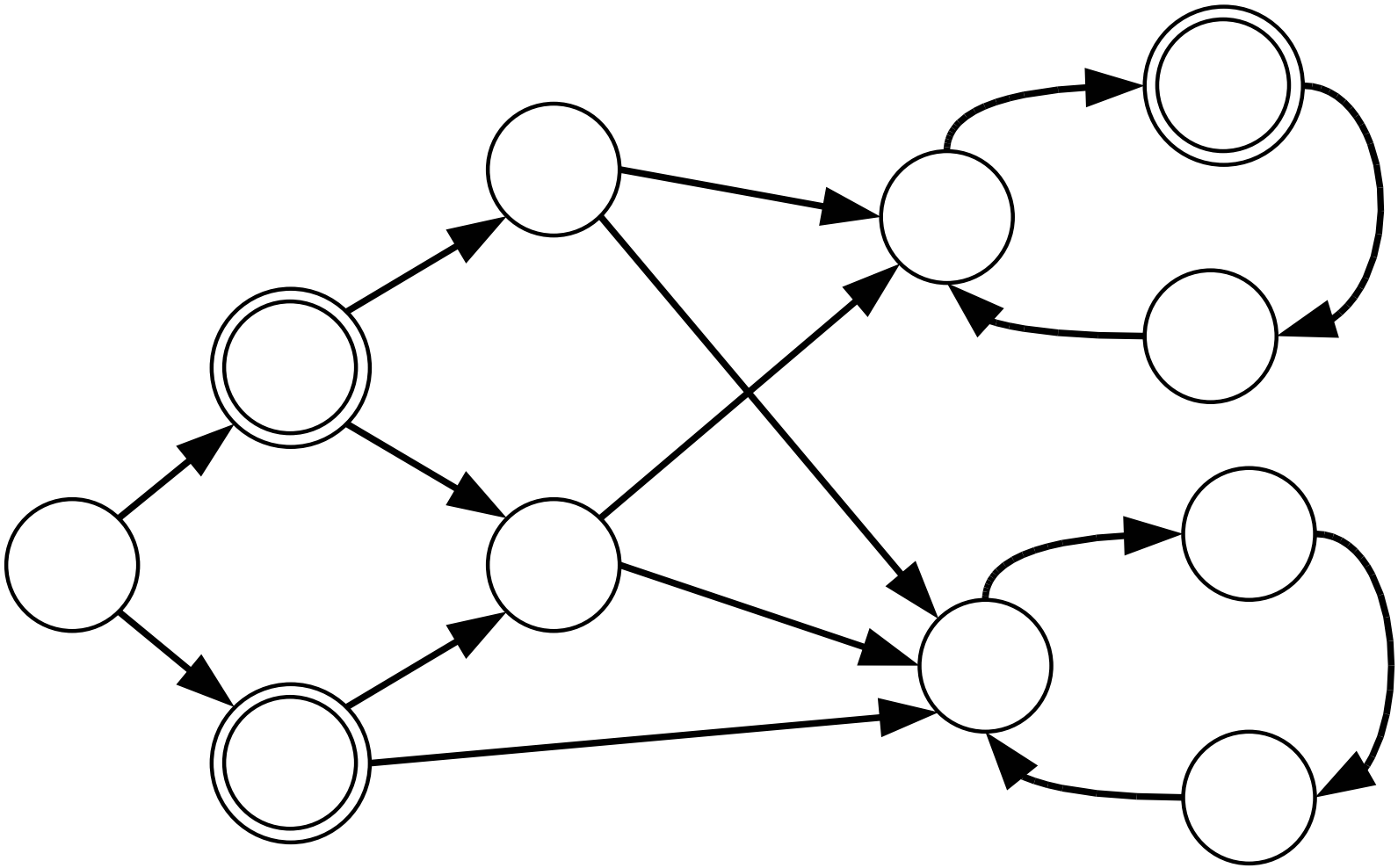
Idea

- Pokud je nějaký akceptující vrchol svým předchůdcem, pak existuje akceptující cyklus
- Vypočítat a uchovávat všechny předchůdce daného vrcholu je nákladné. Množinu všech předchůdců budeme reprezentovat pouze největším (v nějakém uspořádání) vrcholem z dané množiny.
- Pokud je akceptující vrchol svým maximálním předchůdcem, leží na akceptujícím cyklu.

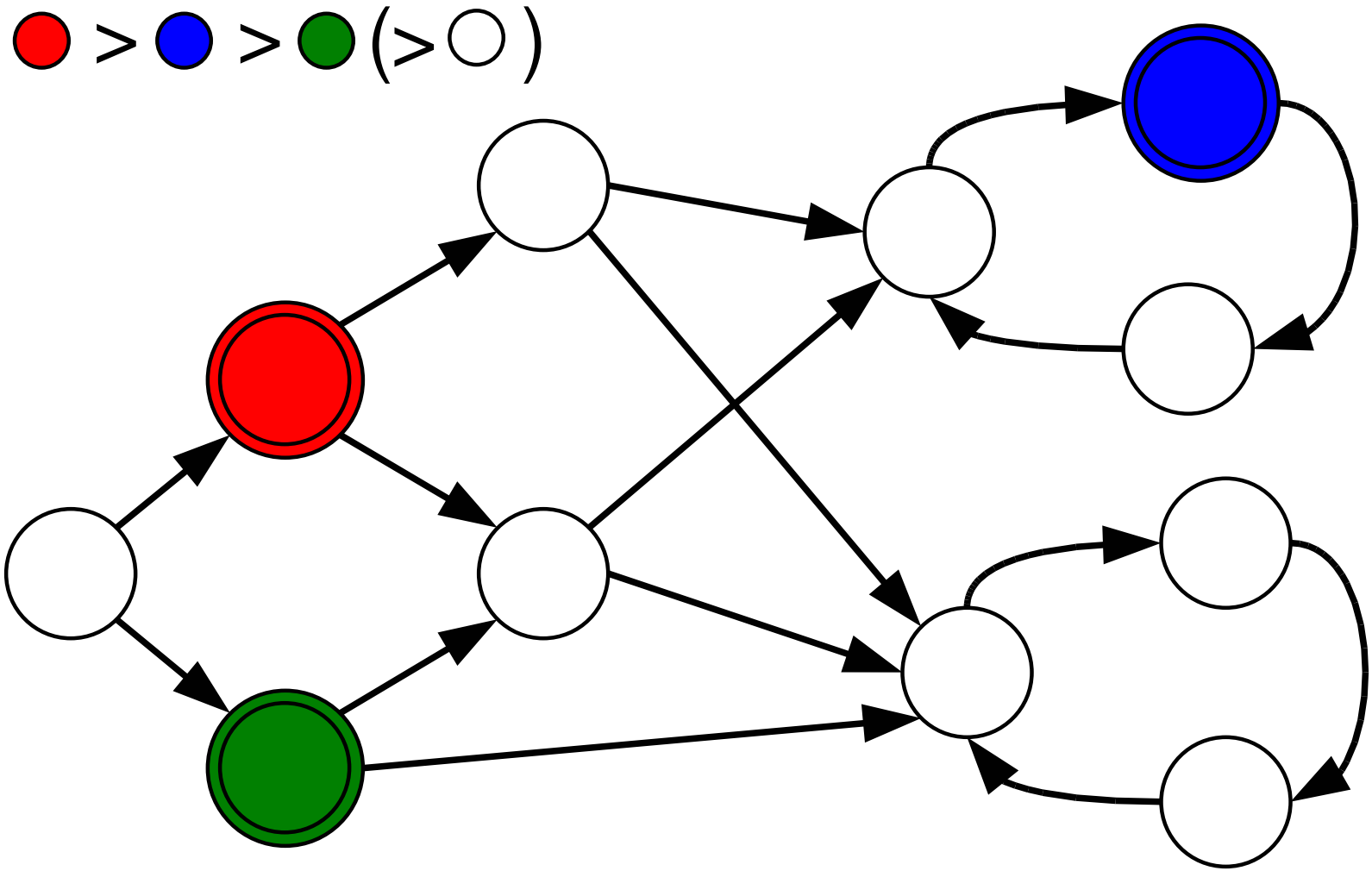
Realizace

- Propagace maximálních akceptujících předchůdců (MAP).
- Pokud je vrchol propagován do sebe sama, je nalezen akceptující cyklus.
- Pokud se to nestane, odstraní se iniciální akceptující vrcholy.
- Propagaci je možné provádět paralelně.

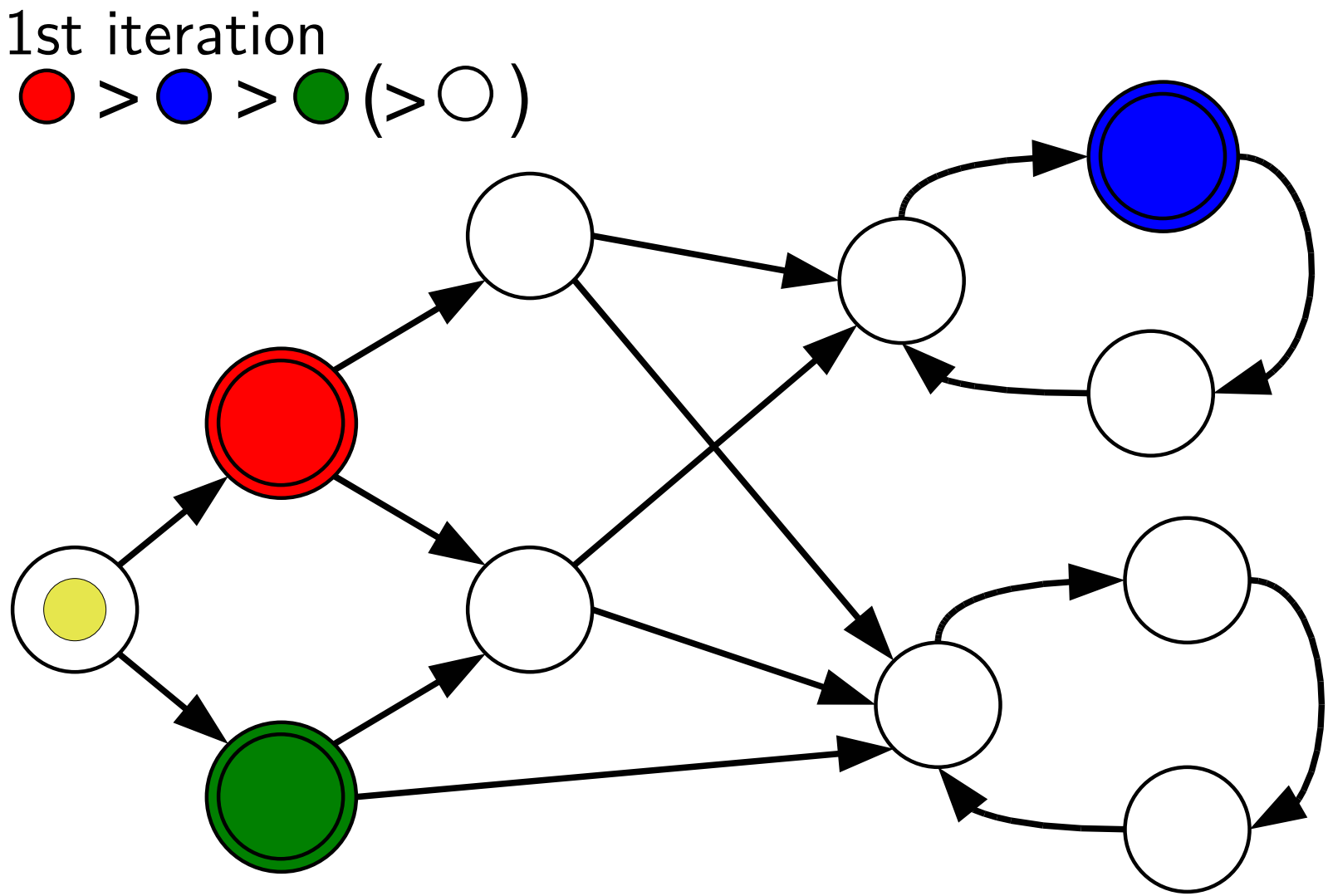
MAP



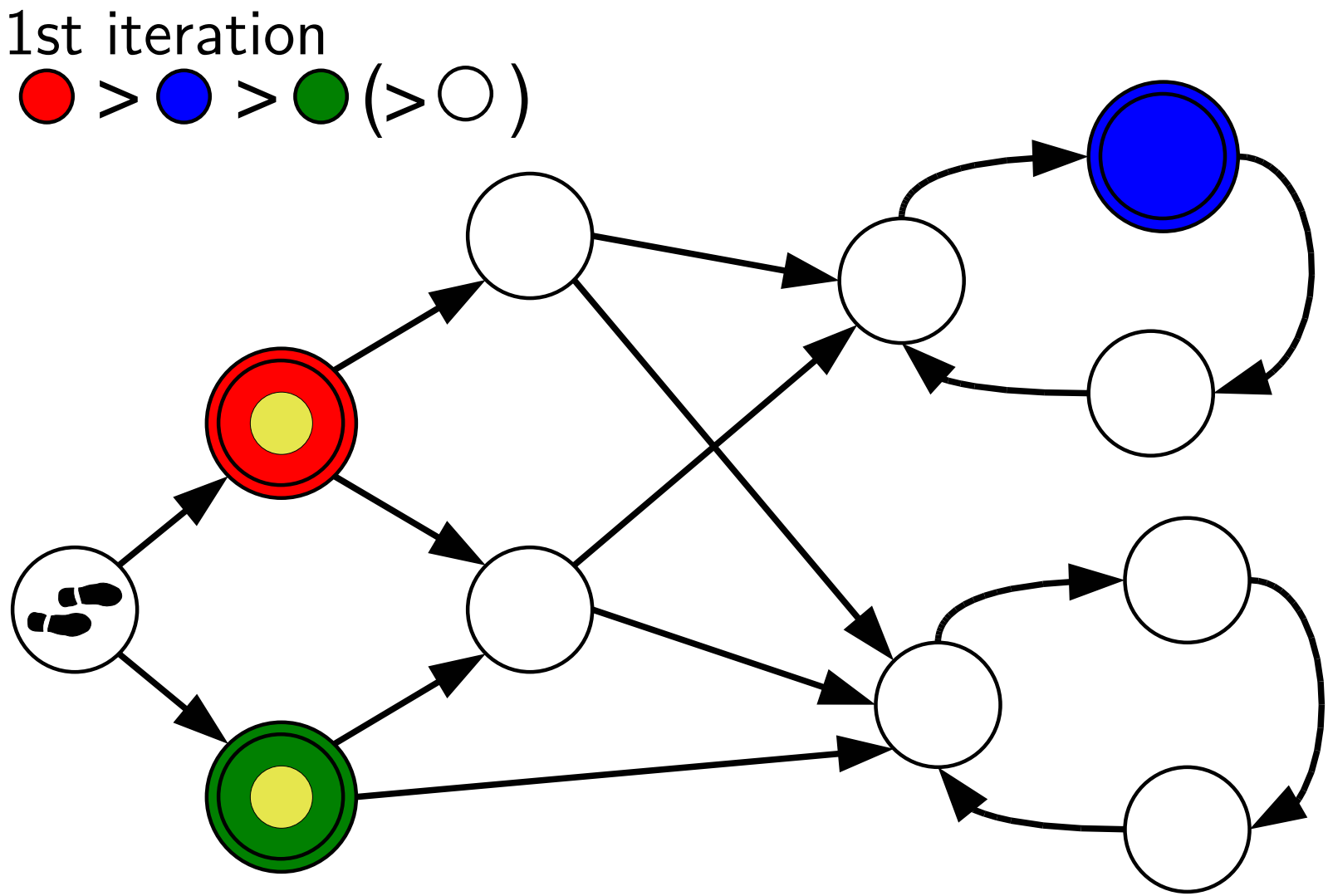
MAP



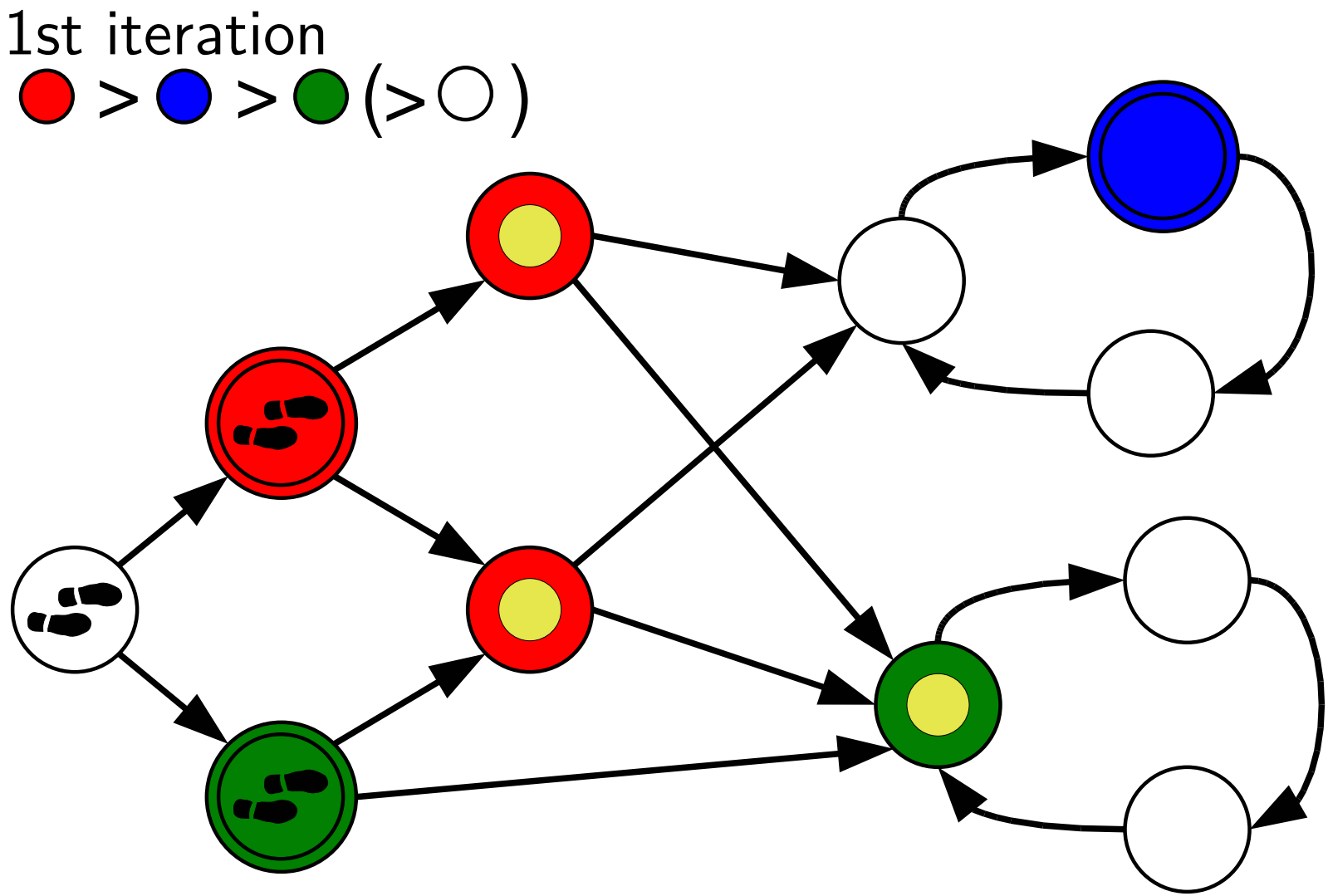
MAP



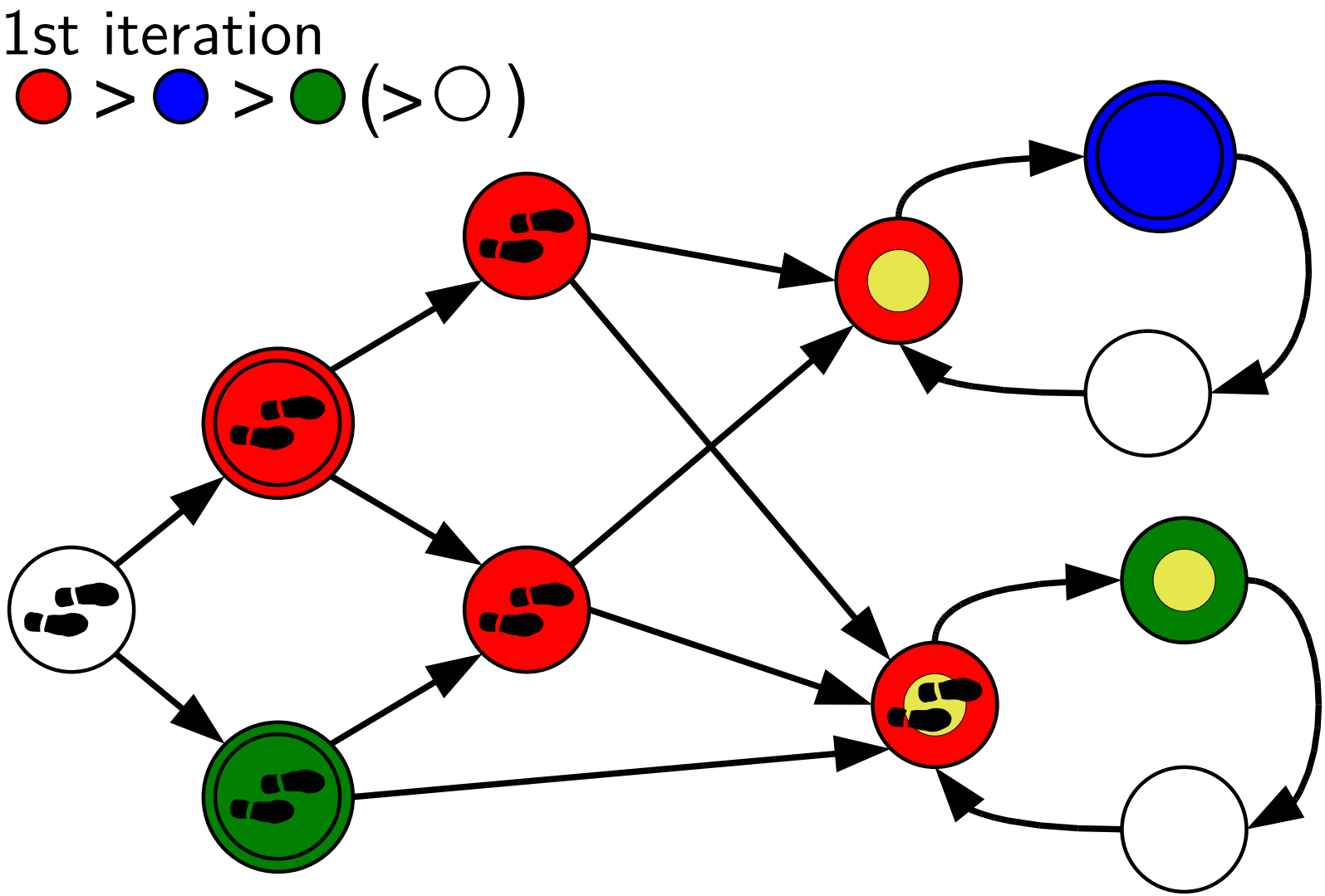
MAP



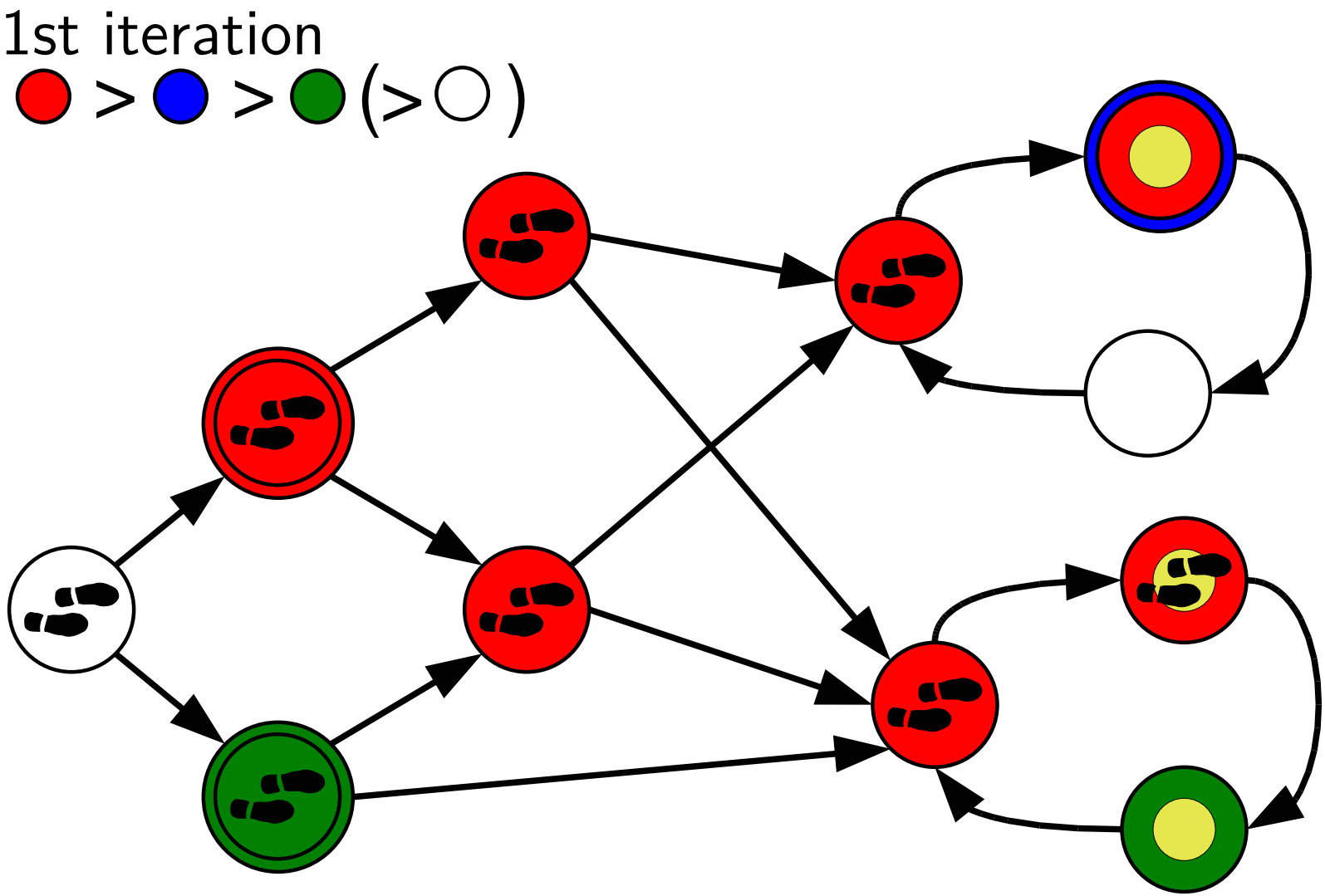
MAP



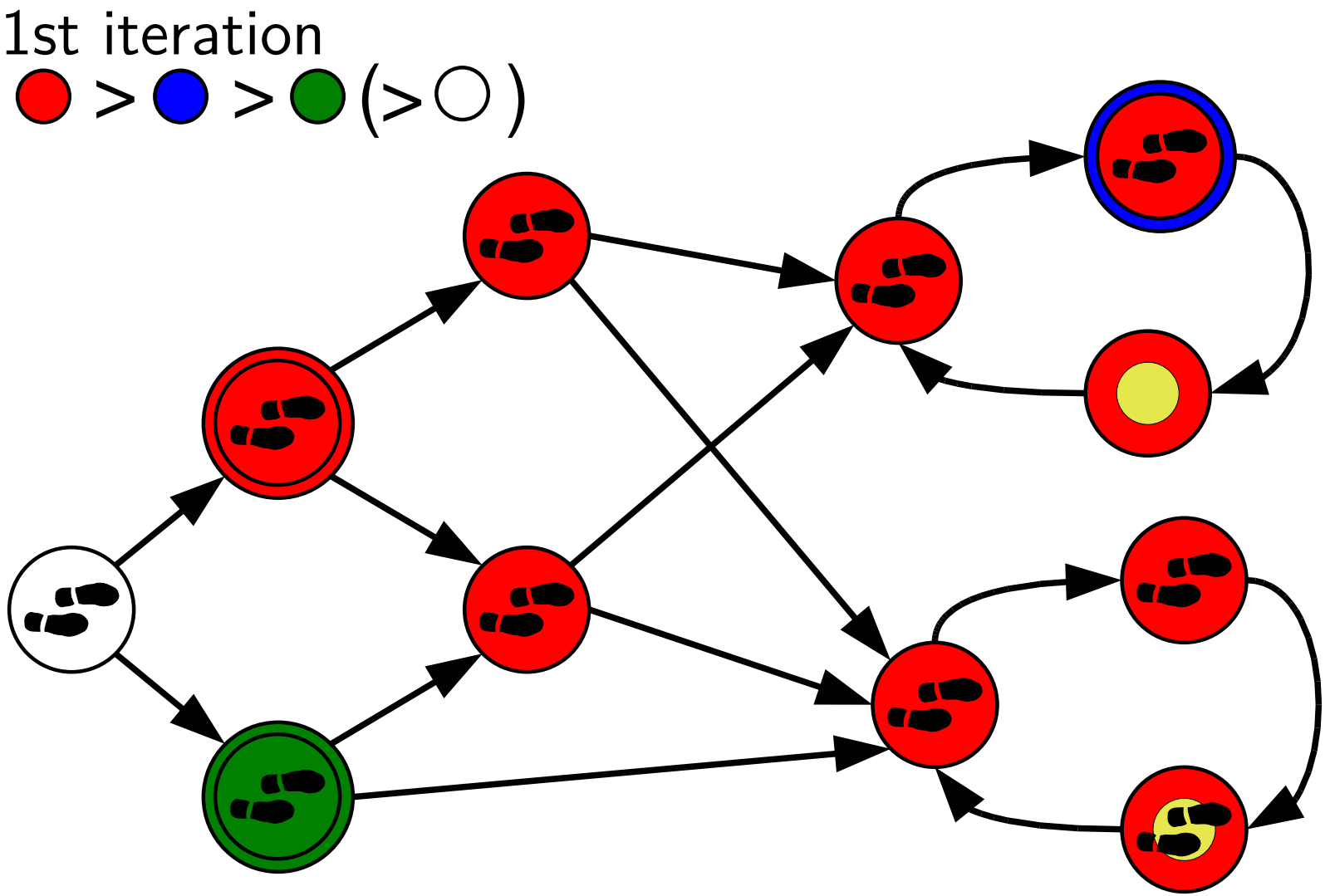
MAP



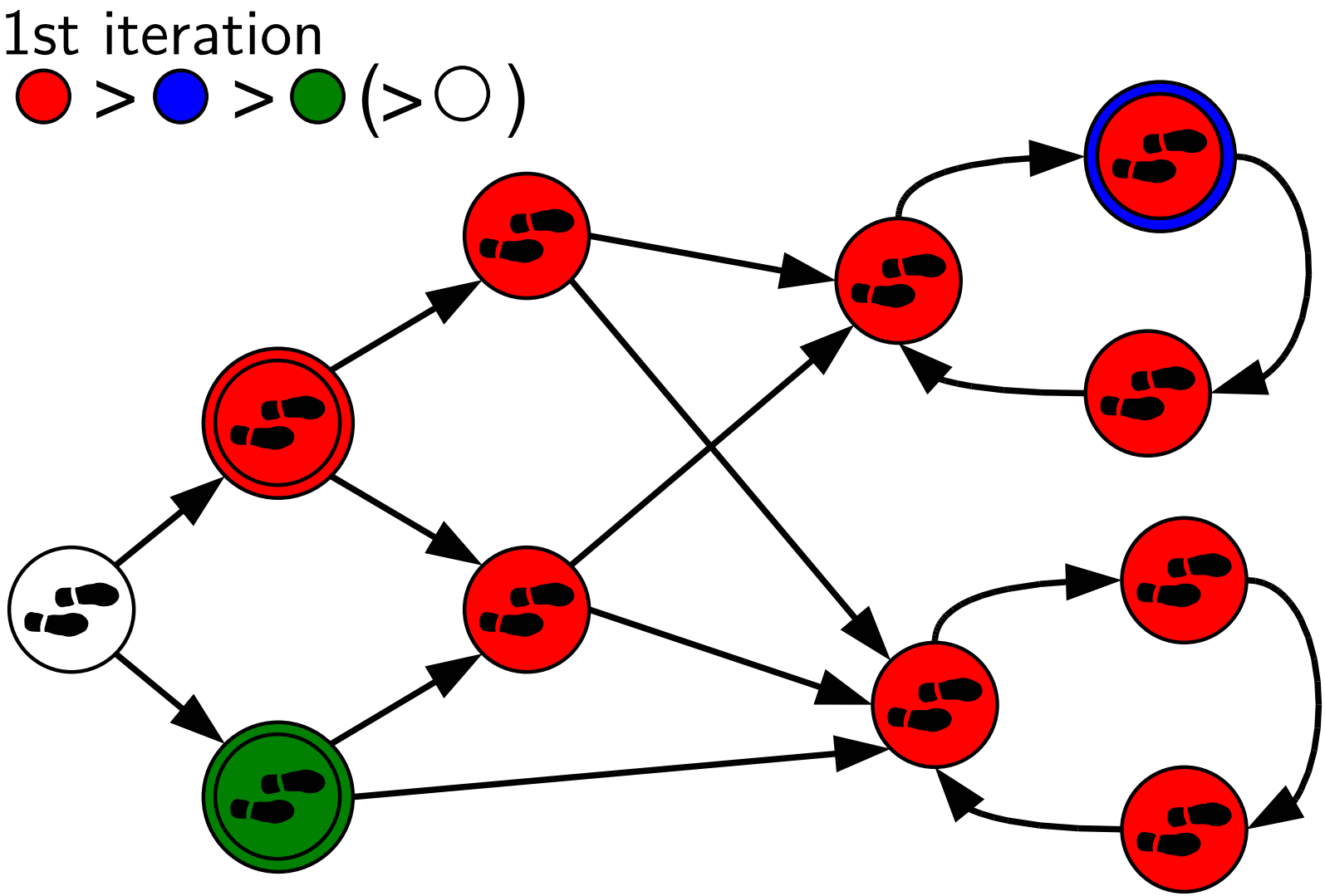
MAP



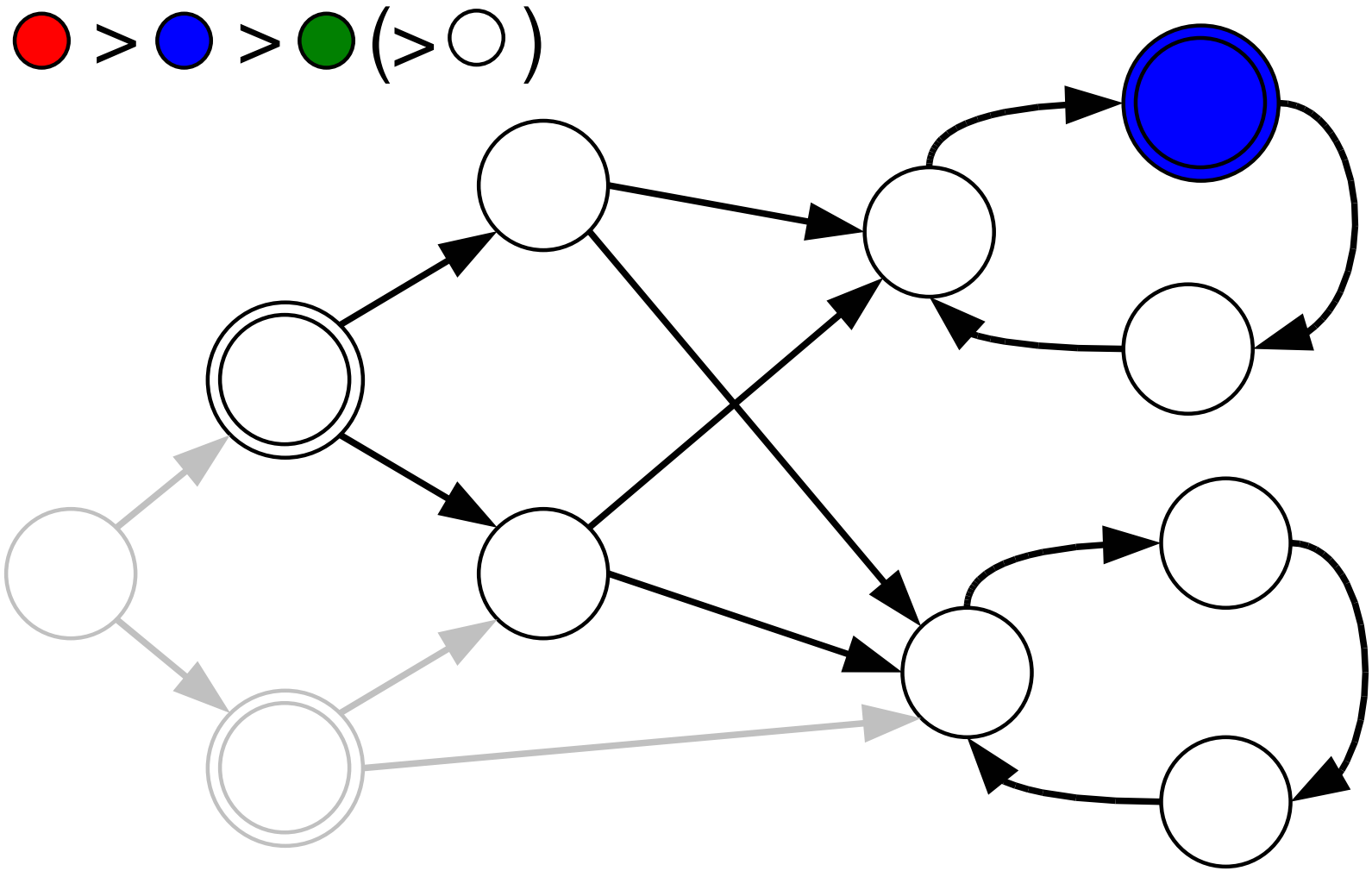
MAP



MAP

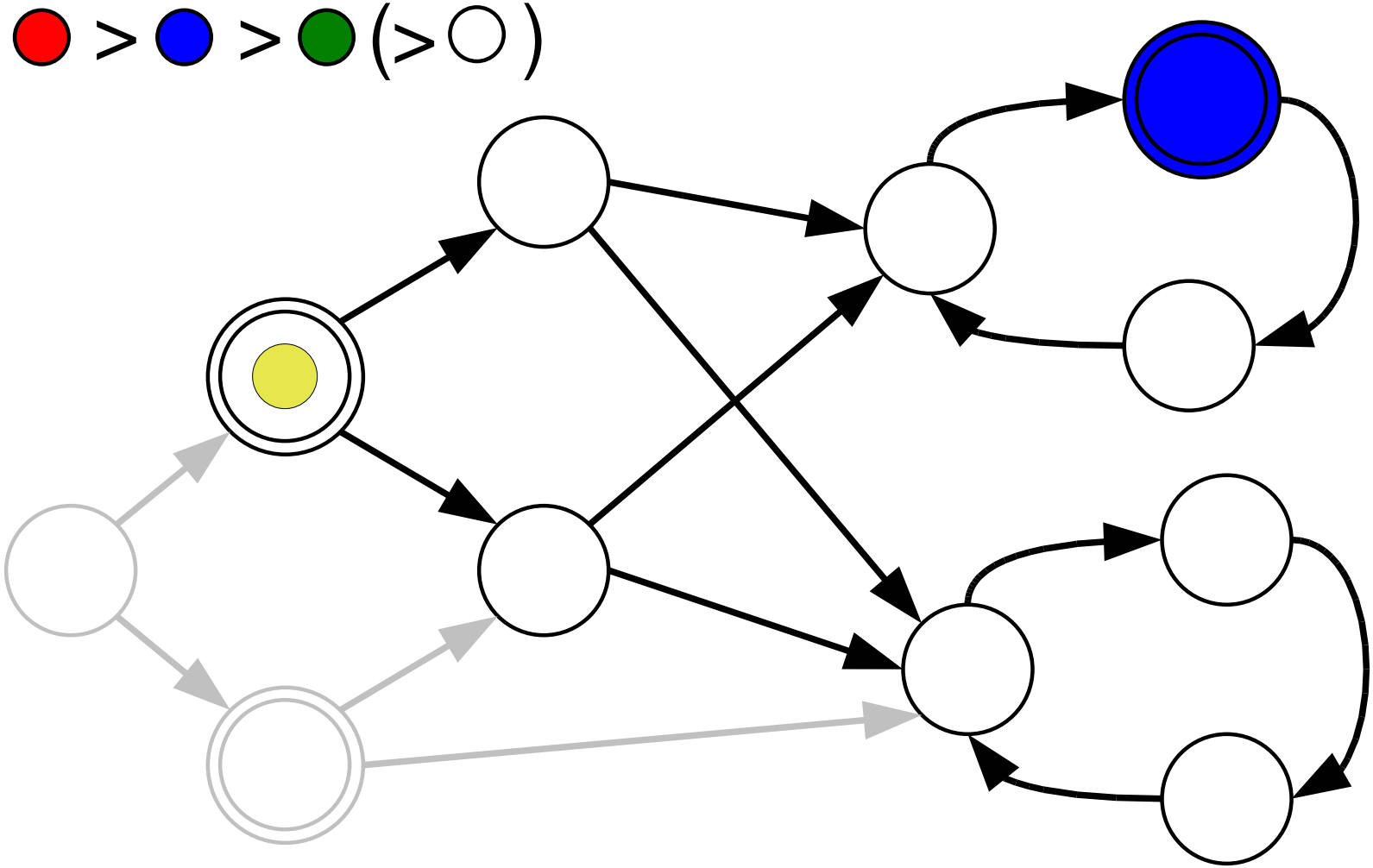


MAP

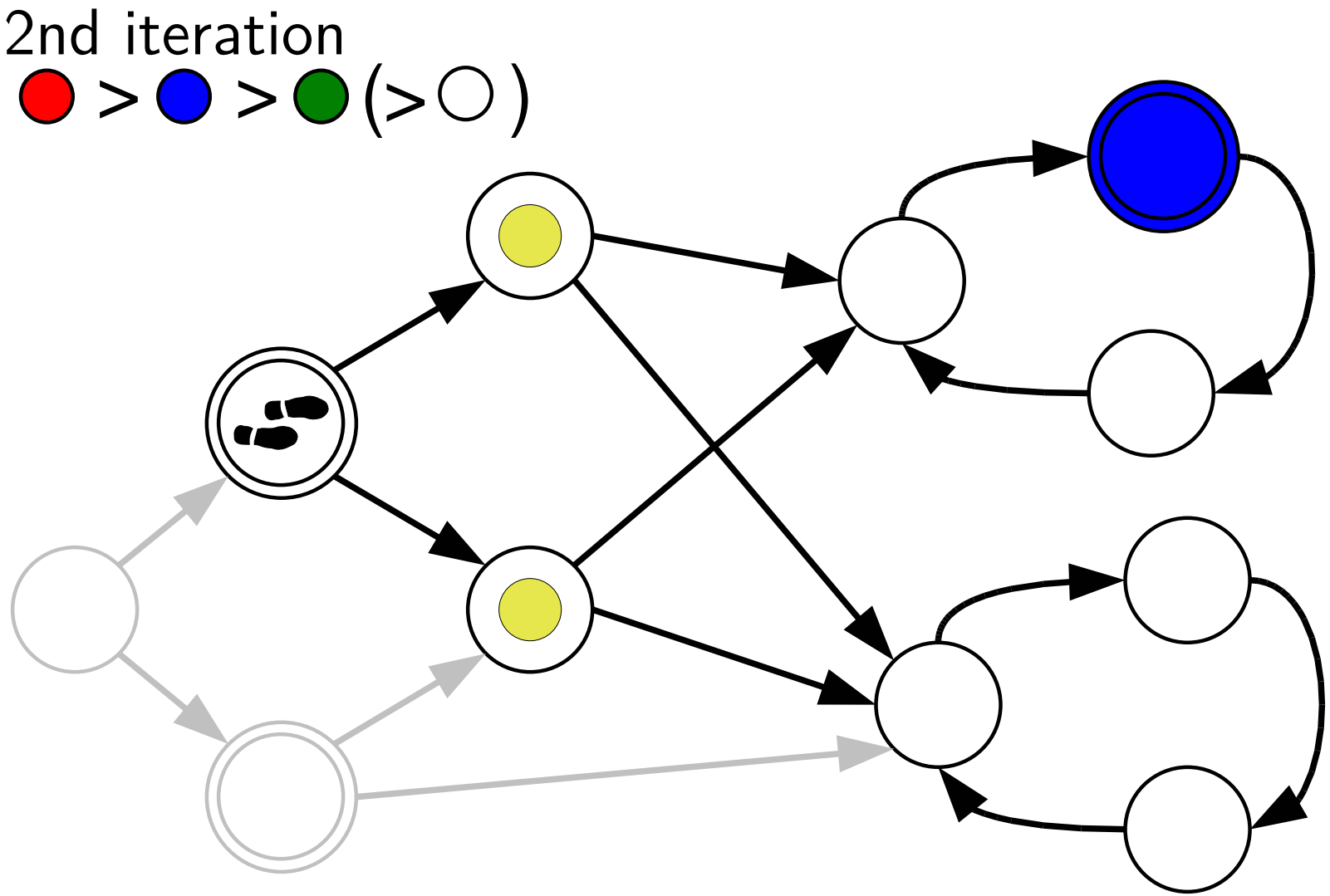


MAP

2nd iteration

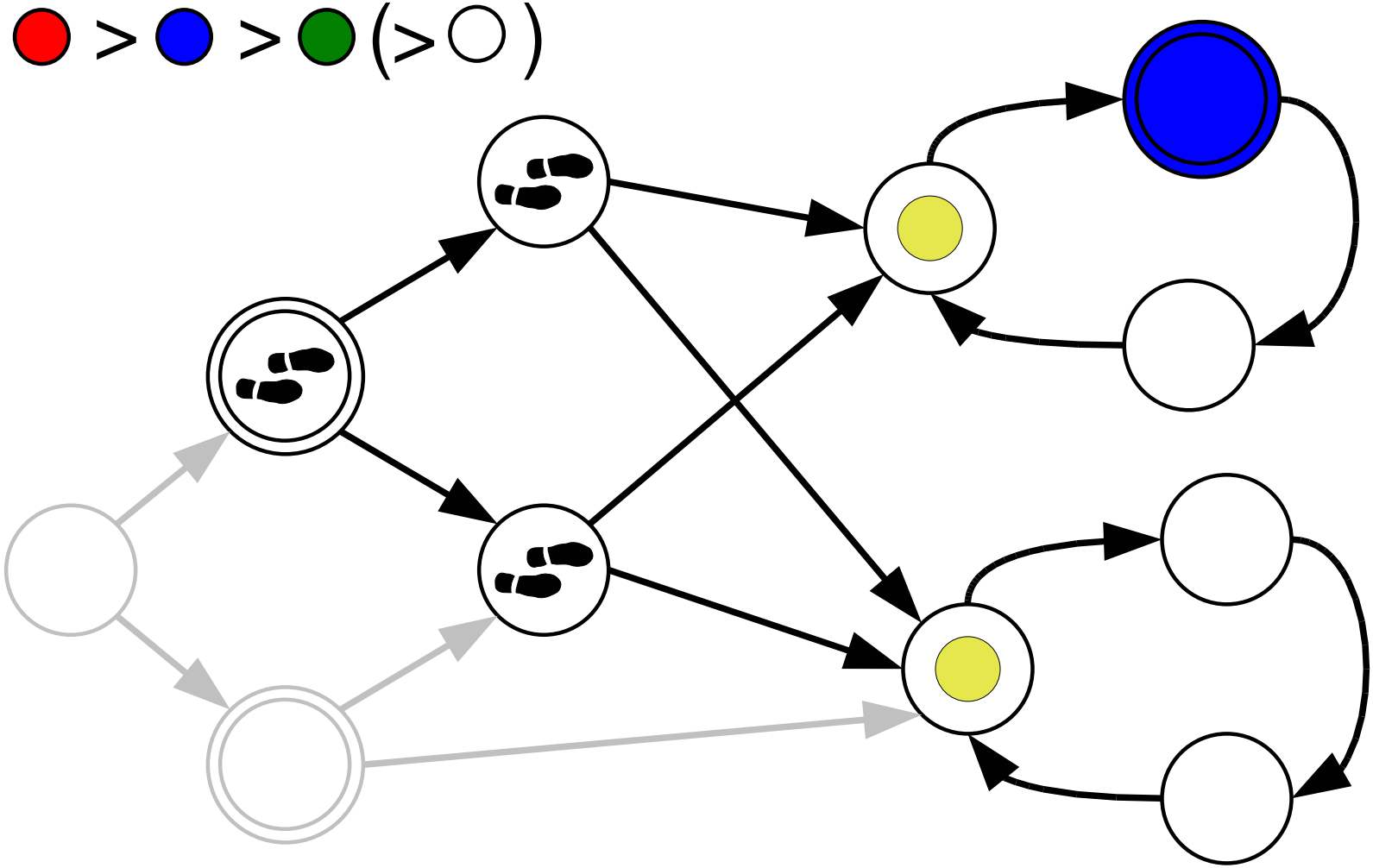


MAP



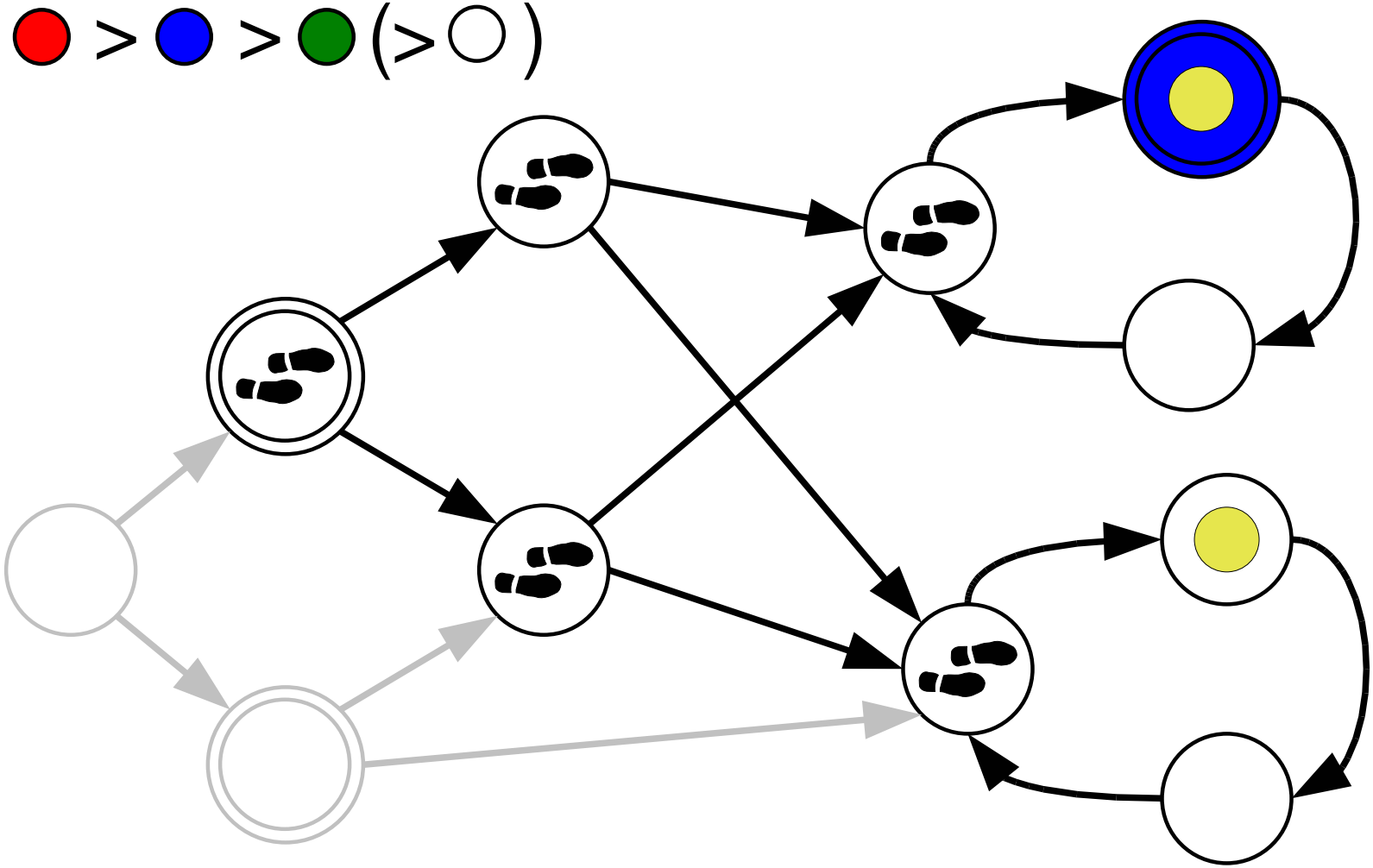
MAP

2nd iteration



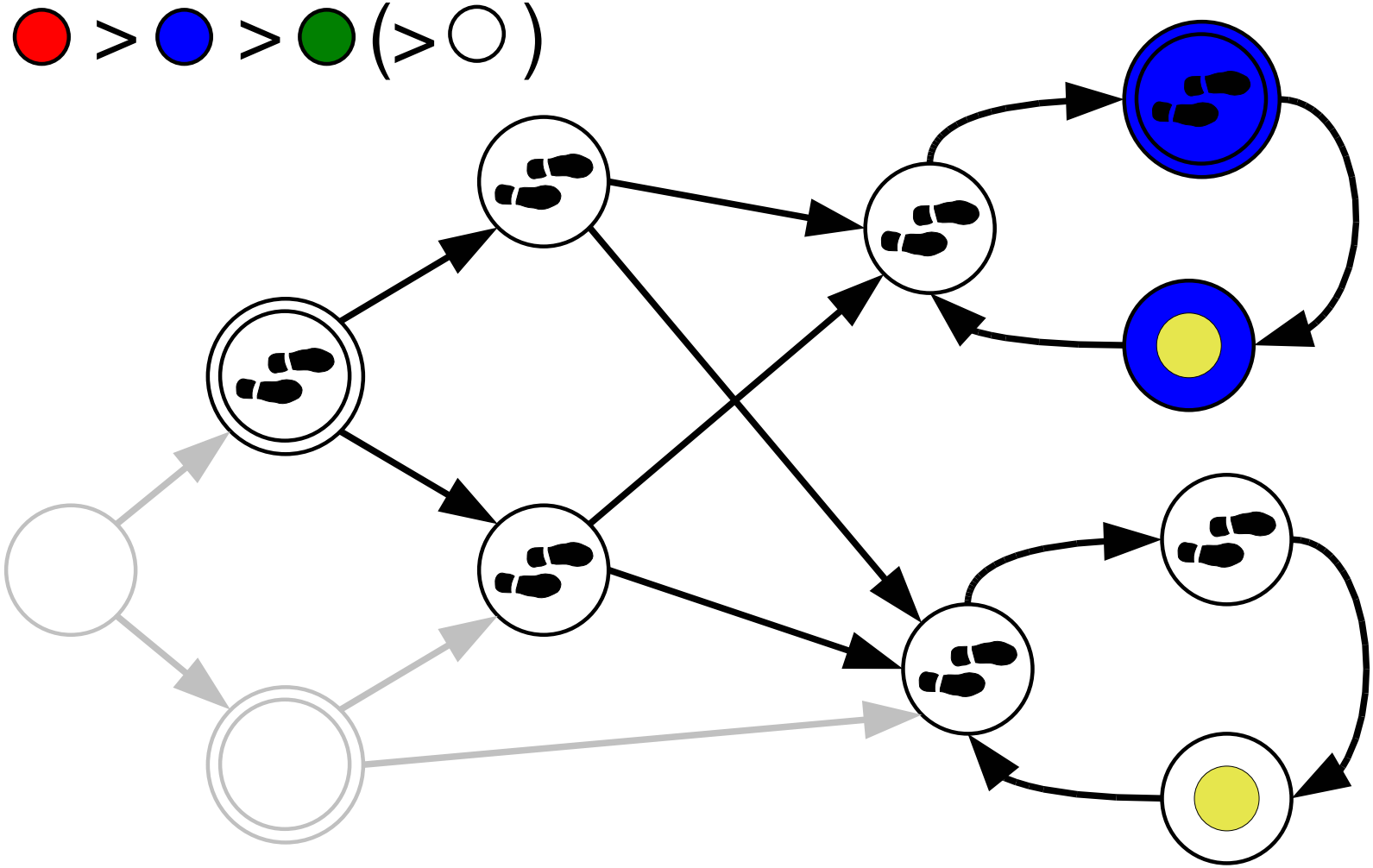
MAP

2nd iteration



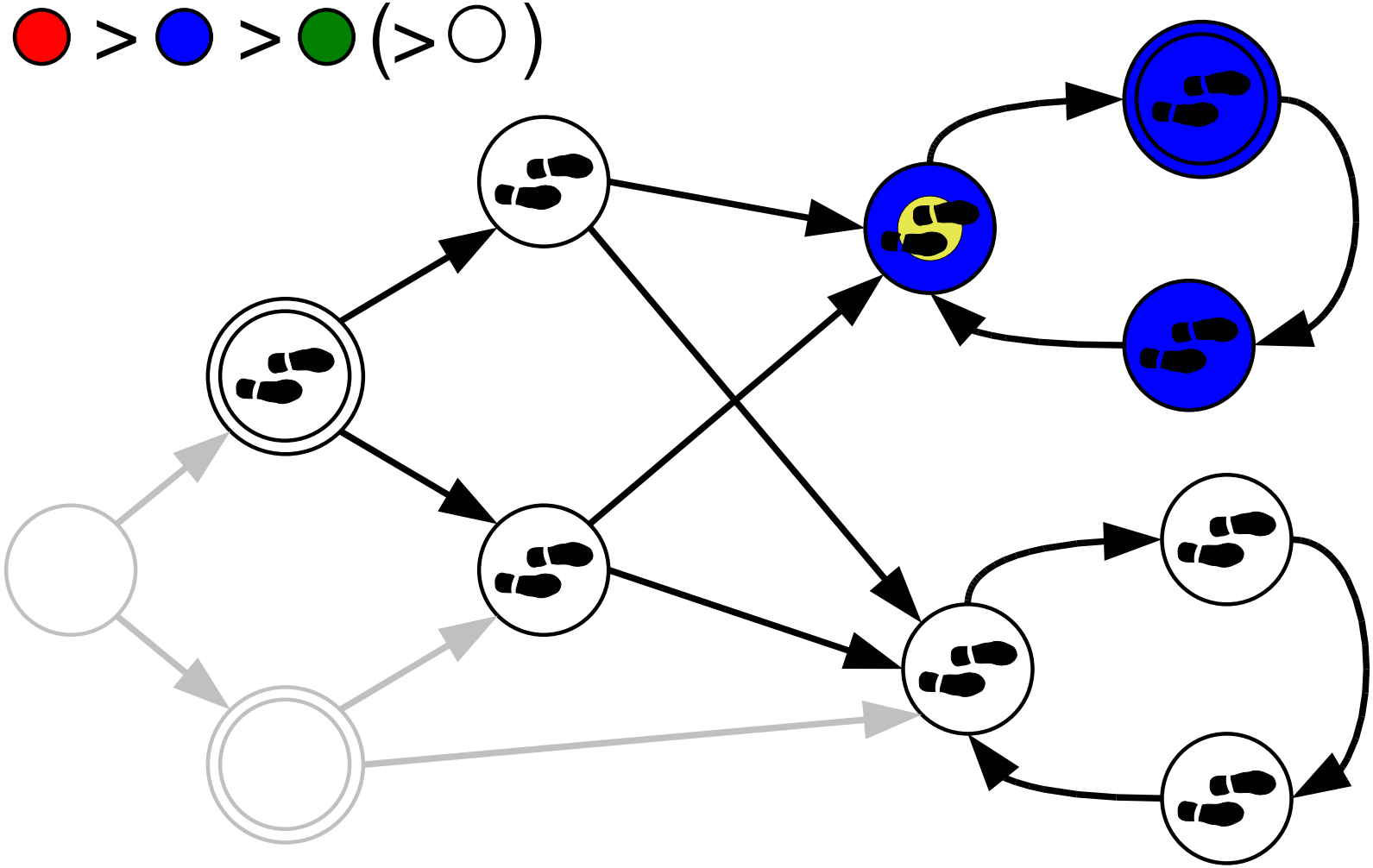
MAP

2nd iteration

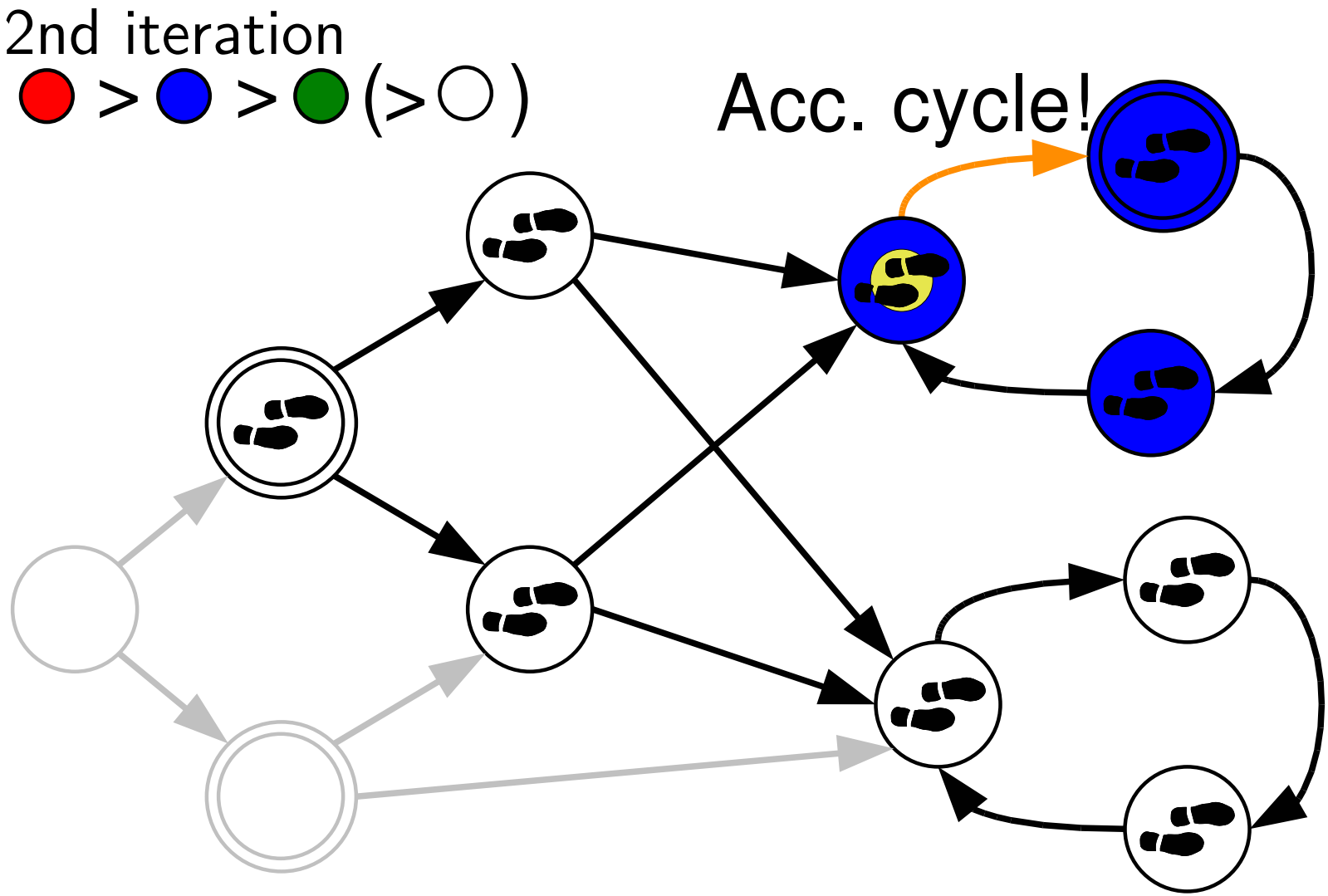


MAP

2nd iteration



MAP



Porovnání algoritmů

	Složitost	Optimalita	Časná detekce
Nested DFS	O(N+M)	Ano	Ano
OWCTY Algorithm	O(N.(N+M))	Ne	Ne
MAP Algorithm	O(N.N.(N+M))	Ne	Ano

N – počet vrcholů
M – počet hran

Možný první dojem: **Nové algoritmy jsou k ničemu.**

Co bychom očekávali při použití paralelního HW

Systémy se sdílenou pamětí

- Redukce času potřebného k detekci akceptujícího cyklu.

Systémy s distribuovanou pamětí (klastry, gridy)

- Agregovaná paměť by měla umožnit analýzu větších grafů.
- Redukce času potřebného k detekci akceptujícího cyklu.

Systémy se sdílenou pamětí

Použitý Hardware

- 16-core server (8x AMD dual core)
- 64 GB RAM

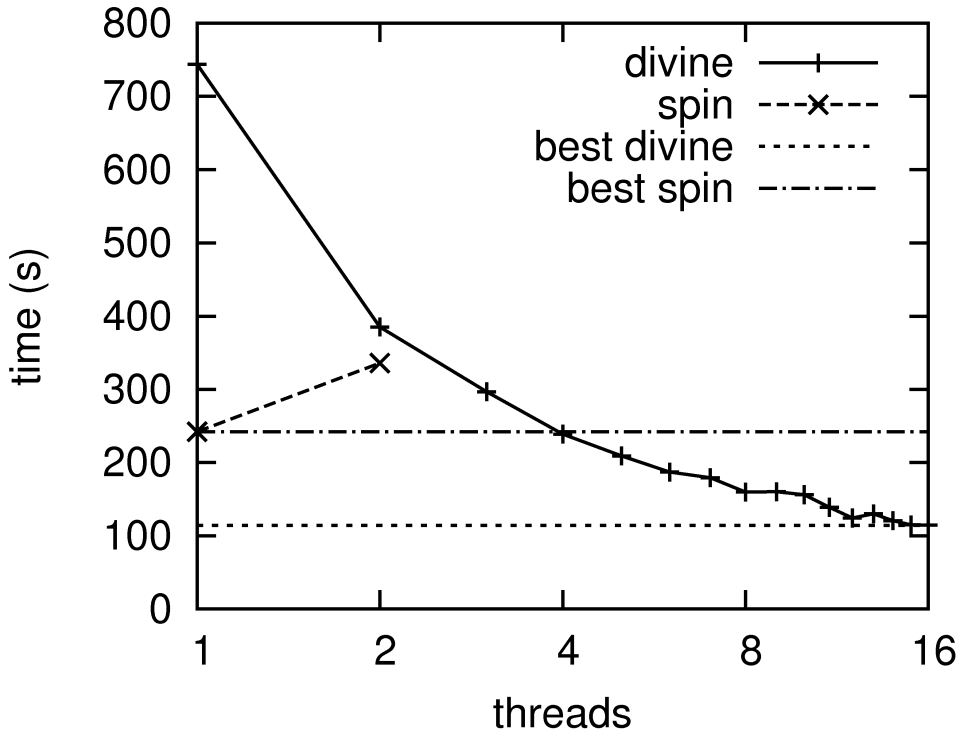
Algoritmy/Nástroje

- SPIN
  - Nested DFS
  - Implementace optimalizována téměř 20 let
  - Znám pro svou obrovskou rychlost
- **DiVinE**
  - Algoritmus OWCTY
  - Realizace implementace: implementace studenty FI

Otázka

- Může neoptimální OWCTY algoritmus v praxi porazit optimální Nested DFS?

Škálovatelnost OWCTY



SPIN: 1 Core: 238 sec
2 Cores: 343 sec

DiVinE:

Cores	Runtime (sec)	Efficiency
1	738	100%
2	392	94%
4	235	78%
8	170	54%
16	106	43%

Systémy se sdílenou pamětí

Otázka

- Může neoptimální OWCTY algoritmus v praxi porazit optimální Nested DFS?

Otázka

- Může neoptimální OWCTY algoritmus v praxi porazit optimální Nested DFS? **Ano**

Systémy s distribuovanou pamětí

Použitý Hardware

- Klastr 64 výpočetních uzlů
- Každý uzel má 2x dual-core CPU a 4 GB RAM
- Myri-10G síťové spojení

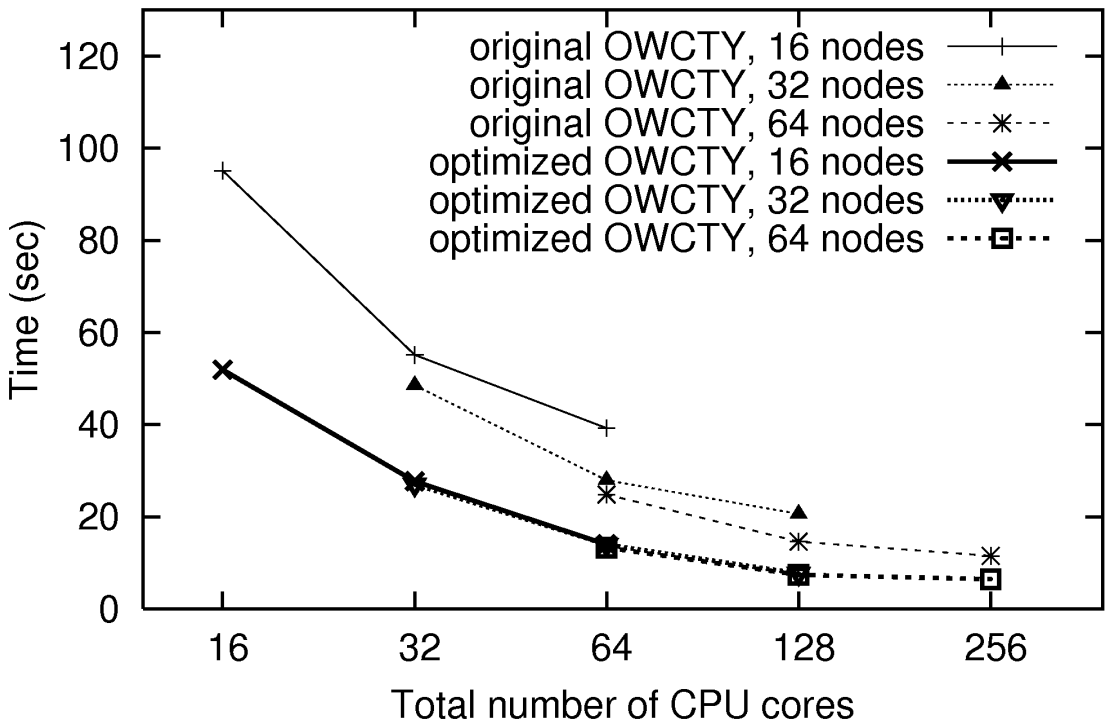
DiVinE Cluster

- Algoritmy OWCTY a MAP
- Síťová vrstva vyladěna expertem na klastrové počítání

Otázky

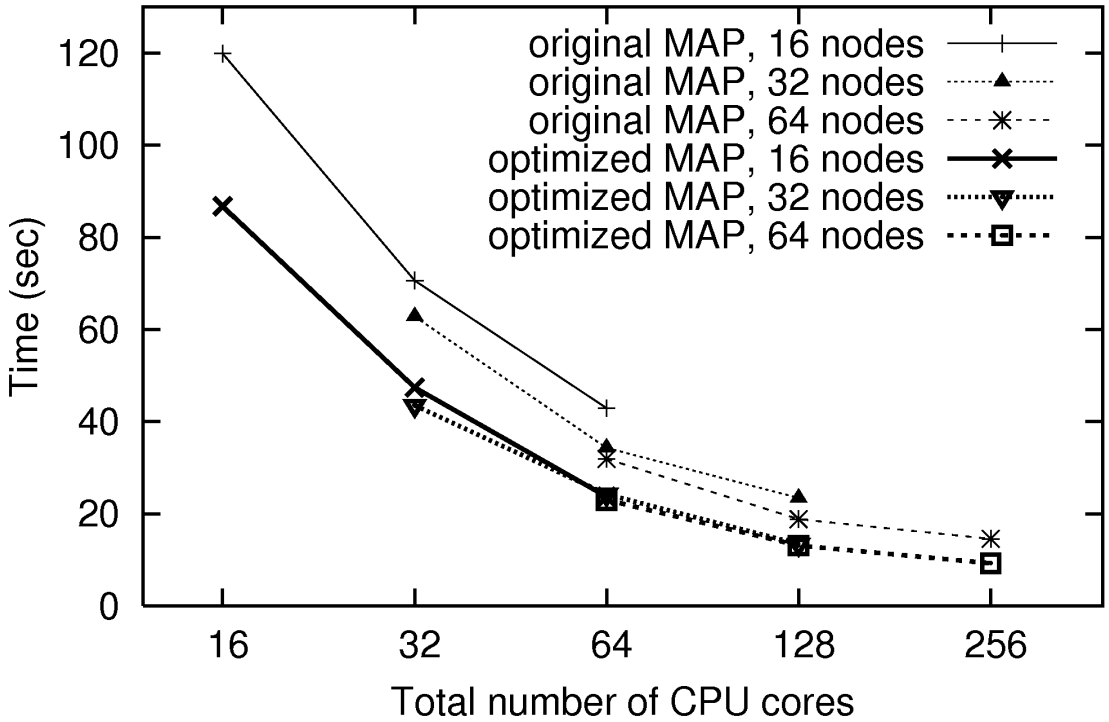
- Je síťová komunikace úzkým místem?
- Může klastr porazit systémy se sdílenou pamětí?
- Je možné analyzovat extrémně velké grafy?

Škálovatelnost OWCTY



Cores	Runtime (sec)	Efektivita
1	631.7	100%
64	13.3	74%
128	7.4	67%
256	5.0	49%

Škálovatelnost MAPu



Cores	Runtime (sec)	Efektivita
1	1052.5	100%
64	23.0	72%
128	13.1	63%
256	8.9	46%

Systémy s distribuovanou pamětí

Otázky

- Je síťová komunikace úzkým místem?
- Může klastr porazit systémy se sdílenou pamětí?
- Je možné efektivně analyzovat extrémně velké grafy?

Systémy s distribuovanou pamětí

Otázky

- Je síťová komunikace úzkým místem? **Ne nutně**
- Může klastr porazit systémy se sdílenou pamětí?
- Je možné efektivně analyzovat extrémně velké grafy?

Systémy s distribuovanou pamětí

Otázky

- Je síťová komunikace úzkým místem? **Ne nutně**
- Může klastr porazit systémy se sdílenou pamětí? **Ano**
- Je možné efektivně analyzovat extrémně velké grafy?

Systémy s distribuovanou pamětí

Otázky

- Je síťová komunikace úzkým místem? **Ne nutně**
- Může klastr porazit systémy se sdílenou pamětí? **Ano**
- Je možné efektivně analyzovat extrémně velké grafy? **Ano**

Závěr

Návrh paralelních algoritmů ač složitostně neoptimálních je nedílnou a důležitou částí práce programátora paralelních aplikací.

Shrnutí

Co jsme se dozvěděli

Paralelní HW platformy

- Principy fungování HW systémů se sdílenou pamětí.
- Komunikace v systémech s distribuovanou pamětí.

Nástroje paralelního programování

- Standardy a knihovny pro implementaci paralelních aplikací.

  `POSIX Threads, OpenMP, MPI (OpenMPI)`
- Principy fungování těchto knihoven

  `Lock-Free datové struktury, Kolektivní komunikace`

Algoritmizace

- Principy návrhu paralelních algoritmů.
- Ukázky paralelních řešení grafových problémů.

Sekce

Shrnutí

Co jsme se dozvěděli

Paralelní HW platformy

- Principy fungování HW systémů se sdílenou pamětí.
- Komunikace v systémech s distribuovanou pamětí.

Nástroje paralelního programování

- Standardy a knihovny pro implementaci paralelních aplikací.
  `POSIX Threads, OpenMP, MPI (OpenMPI)`
- Principy fungování těchto knihoven
  `Lock-Free datové struktury, Kolektivní komunikace`

Algoritmizace

- Principy návrhu paralelních algoritmů.
- Ukázky paralelních řešení grafových problémů.