Základy paralelizácie

Už v úvode sme sa zmienili, že súčasný trend vyžaduje spracovanie čoraz väčšieho množstva údajov a používanie komplexnejších algoritmov. Keďže potrebný nárast výpočtového výkonu nie je možné garantovať neustálym zvyšovaním výkonu jednojadrového procesora alebo optimalizáciou zdrojového kódu, ako schodné riešenie sa ponúka použitie paralelizácie programového vybavenia. Túto myšlienku podporuje aj fakt, že v súčasnosti väčšina dostupného hardvéru počnúc od mobilných zariadení cez notebooky a desktopy až po servre, počítačové klastre a superpočítače je vybavených viacjadrovými procesormi. Naopak softvér v tejto oblasti zaostáva. Väčšina bežne používaného dostupného softvéru s výnimkou operačného systému, niektorých serverových služieb a špeciálneho vedeckého softvéru stále nedokáže plne využiť potenciál tohto paralelného hardvéru.

V tejto kapitole si objasníme základné princípy a postupy, ktoré sa uplatňujú pri tvorbe paralelných programov. Základným princípom paralelizácie je vhodne rozdeliť výpočtovo náročnú úlohu na viacero podúloh, ktoré je možné riešiť viac-menej samostatne. Vhodnou kombináciou výsledkov týchto podúloh môžme následne získať celkový výsledok. V prípade, keď náš paralelný program budeme používať iba v rámci jedného uzla, či už sa jedná o desktop alebo výpočtový uzol klastra, spravidla využívame model paralelného programu so zdieľanou pamäťou. V takomto prípade sa náš program spúšťa ako jeden proces s jedným vláknom a počas behu programu sa v rámci procesu vytvorí niekoľko ďalších vlákien riešiacich príslušné podúlohy. V prípade použitia viacerých výpočtových uzlov je nevyhnutné použiť model paralelného programu s distribuovanou pamäťou. Takýto program sa spúšťa v podobe niekoľkých samostatných procesov, ktoré navzájom môžu komunikovať. Model s distribuovanou pamäťou je tiež možné využiť aj v paralelných programoch, ktoré budú spúšťané v rámci jedného uzla. Všetky zmienené skutočnosti v tejto kapitole je možné využiť v oboch modeloch, ako so zdieľanou, tak aj s distribuovanou pamäťou.

Väčšina začínajúcich používateľov paralelných programov sa chybne domnieva, že čím viac výpočtového výkonu (viac procesorov) bude určitú úlohu riešiť, tým bude program rýchlejší. Tento predpoklad závisí od mnohých faktorov, ako napríklad charakter samotnej riešenej úlohy, spôsobu dekompozície problému, množstva vzájomnej komunikácie a synchronizácie v podúlohách a mnohých ďalších. Z tohto dôvodu dochádza často k mrhaniu veľkého množstva výpočtového výkonu, nakoľko nie všetky paralelné programy dokážu bežať efektívne na príliš veľkom počte procesorov. Preto aj voľba vhodného paralelného hardvéru a jeho množstva zohráva kľúčovú úlohu pre efektívne využívanie prostriedkov pre vysokovýkonné počítanie.

V súčasnosti je veľké množstvo používateľov aj z vedeckej sféry odkázaných na používanie sériových programov, pričom mnohým z

nich výkon týchto programov aj postačuje. K písaniu a používaniu paralelných programov sa zväčša prikláňame z dvoch dôvodov:

- keď výpočtový výkon sériového programu nie je postačujúci na vyriešenie danej úlohy v akceptovateľnom čase (môže byť chápané aj ako niekoľko hodín, či dní),
- keď je pamäťová náročnosť programu väčšia, ako je množstvo pamäte v jednom výpočtovom uzle.

Prvý z uvedených dôvodov sa vyskytuje oveľa častejšie v dôsledku nástupu paralelných architektúr. Druhý z dôvodov bolo v minulosti možné riešiť odkladaním určitej časti údajov mimo hlavnú pamäť počítača na sekundárne úložisko. Súčasné paralelné počítače disponujú úložiskami s vysokou rýchlosťou v podobe rôznych paralelných súborových systémov. Ich benefity je však možné plne využiť len použitím paralelného prístupu k uloženým údajom.

1 Paralelizmus

Pred samotným napísaním paralelného programu je nevyhnutné analyzovať použitý algoritmus riešenia problému a identifikovať **inherentný paralelizmus** tomuto algoritmu. Jedná sa o paralelizmus, ktorý je vlastný danému algoritmu alebo problému. Pre rôzne metódy (algoritmy) riešenia toho istého problému môže byť vlastný rôzny stupeň inherentného paralelizmu. Až po jeho identifikovaní je možné vhodne vybrať metódu paralelizácie. Viac informácií a prehľad vzorov paralelného programovania je možné nájsť v literatúre [1].

Postup návrhu paralelného programu zväčša prebieha v nasledujúcich krokoch [2, 3]:

- dekompozícia výpočtového problému na podúlohy, ktoré je možné vykonávať súbežne a návrh sekvenčného algoritmu týchto podúloh,
- analýza granularity výpočtov,
- minimalizácia nákladov paralelného algoritmu,
- priradenie podúloh jednotlivým procesom vykonávajúcim paralelný algoritmus.

Dekompozícia problému

Na základe identifikácie inherentného paralelizmu algoritmu je problém možné rozložiť na niekoľko podproblémov, ktoré je možné riešiť nezávisle na sebe. Hovoríme o **dekompozícií problému**. V tejto kapitole si opíšeme niekoľko spôsobov dekompozície paralelného problému na príkladoch.

Pre lepšiu ilustráciu si môžme predstaviť, že potrebujeme opraviť n testov študentov. Každý test obsahuje m otázok, na ktoré študenti odpovedali. Pre rýchlejšie opravenie testov ich môžme rozdeliť viacerým opravujúcim tak, že každý z nich bude opravovať všetky otázky v určitej podskupine z množstva testov. Počet takto vzniknutých podúloh

je rovný počtu testov n, ktoré je potrebné opraviť. Každý opravovateľ vykonáva tú istú činnosť s iným testom, a preto v takomto prípade hovoríme o **údajovej dekompozícií**. Naopak, ak opravovanie testov rozdelíme tak, že každý opravovateľ bude opravovať len jednu z otázok v každom teste, vznikne nám m nezávislých podúloh. To znamená, že každý opravovateľ vykonáva inú činnosť so všetkými testami, vtedy hovoríme o **funkcionálnej dekompozícií**.

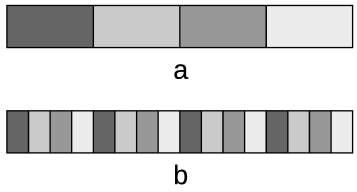
Prv než sa oboznámime s rôznymi modelmi dekompozície, je potrebné podotknúť, že v praxi sú mnohé programy komplexnejšie a pozostávajú z viacerých krokov. V každej časti je pritom možné využiť rôzne modely dekompozície, vtedy hovoríme o tzv. **hybridnej dekompozícií.**

Údajová dekompozícia

Veľmi častým problémom, s ktorým sa v oblasti vysokovýkonného počítania stretávame, je potreba spracovania veľkého množstva údajov. V ideálnom prípade je možné ľubovoľnú časť týchto údajov spracovať nezávisle na iných častiach, a preto je možné tieto časti spracovávať viacerými procesormi súčasne. Takýto prístup označujeme ako **SPMD** (Single Program Multiple Data), čiže ten istý program je vykonávaný na všetkých procesoroch, pričom každý z procesorov pristupuje pomocou ukazovateľa k inej časti údajov uložených v pamäti. Údajová dekompozícia môže byť použitá na dekompozíciu ako vstupných, tak aj výstupných údajov.

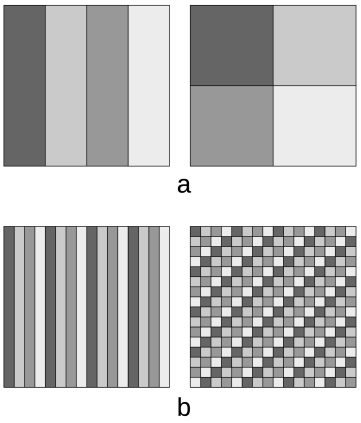
Poznámka: Údajovú dekompozíciu niekedy označujeme aj ako doménovú dekompozíciu.

Vo vedeckých výpočtoch sa bežne stretávame s údajmi uloženými v poliach, pričom prvky poľa sú spracovávané postupne pomocou cyklu. Typickým príkladom sú operácie lineárnej algebry na vektoroch a maticiach, tak ako sú implementované v štandardnej knižnici BLAS (Basic Linear Algebra Subprograms) [4, 5]. V mnohých prípadoch je výpočet pre každý z prvkov poľa nezávislý od ostatných prvkov. Vtedy je možné rozdeliť úlohu na rovnaké menšie podúlohy, pričom každá podúloha má na starosti len určitú časť poľa. Na obrázku 1 sú znázornené dva rôzne spôsoby rozdelenia úlohy na podúlohy, pričom farebne sú odlíšené segmenty spracovávané rôznymi procesormi. Podobne je možné rozdeliť aj spracovanie údajov uložených v maticiach podľa schémy na obrázku 2.



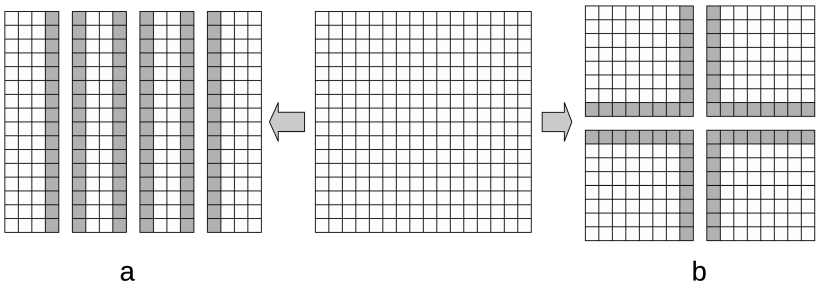
Obr. 1: Rozdelenie poľa na časti (a) po blokoch, (b) cyklicky

Zložitejšia situácia nastáva pri výpočtoch, ako sú simulácie fyzikálnych javov, napríklad tok tekutín, mechanické namáhanie a podobne. Zväčša ide o tak zložité procesy, že je potrebné pristúpiť k určitému zjednodušeniu reality, kde môžme pracovať s výpočtovými doménami. Výpočtová doména môže predstavovať napríklad určité množstvo kvapaliny, ktoré je reprezentované ako mriežka diskrétnych bodov v priestore. Skúmaním toku kvapaliny potom vlastne sledujeme pozíciu týchto bodov, napríklad v Karteziánskom priestore, pričom tieto body sa prispôsobujú na základe numerických podmienok určených použitým algoritmom. Priamočiare riešenie takýchto algoritmov predstavuje **doménová dekompozícia**, čiže rozdelenie mriežky študovaných bodov na určité časti a pridelenie každej časti jednému z procesorov, napríklad podľa schémy znázornenej na obrázku 2 (a).



Obr. 2: Rozdelenie matice na časti (a) po blokoch, (b) cyklicky

Takéto algoritmy predstavujú isté komplikácie v tom, že výpočty jednotlivých domén, ktoré sú vykonávané viacerými procesormi, nie sú počas priebehu jednotlivých iterácií algoritmu na sebe úplne nezávislé. Pri použití architektúry so zdieľanou pamäťou má každý procesor na starosti výpočty v určitom výseku množiny bodov, pričom výsledné hodnoty opätovne aktualizuje v zdieľanej pamäti, takže ostatné procesory v nasledujúcej iterácii majú prístup k potrebným údajom. Pri použití architektúry s distribuovanou pamäťou je situácia zložitejšia v tom, že každý z procesorov potrebuje mať k dispozícii informácie aj o hraničných bodoch zo susedných výsekov. Tieto je preto potrebné po každej iterácii komunikovať medzi procesmi vykonávajúce výpočty v susediacich doménach. Z tohto dôvodu je potrebné zamerať sa na to, aby výpočty v jednotlivých doménach boli dobre vyvážené, pretože v opačnom prípade by dochádzalo k tomu, že jednotlivé procesy musia na seba dlho čakať, čo vedie k nízkej efektivite paralelného programu. V druhom rade sa treba tiež zamerať na minimalizáciu množstva dát, ktoré je po každej iterácii potrebné odkomunikovať medzi procesmi. Ako príklad nám môže poslúžiť dekompozícia dvojrozmerného priestoru bodov s rozmermi $n \times n$ medzi p procesorov tak, ako je znázornená na obrázku 3, pričom sivo označené bloky je potrebné komunikovať.



Obr. 3: Rôzne spôsoby doménovej dekompozície

Úloha: Navrhnite spôsob implementácie paralelného algoritmu na prevod obrázku z odtieňov sivej na čierno-biely obraz metódou prahovania.

Úloha: Navrhnite spôsob implementácie paralelného algoritmu Gaussovho filtra aplikovaného na obrázok v odtieňoch sivej pre architektúru so zdieľanou pamäťou a pre architektúru s distribuovanou pamäťou s použitím údajovej dekompozície.

V prvom prípade (obrázok 3 (a)), bude množstvo údajov, ktoré je potrebné komunikovať škálovať s $O(n(p-1))$. Zatiaľ čo v druhom prípade (obrázok 3 (b)), bude potrebná komunikácia škálovať s $O(2n(\sqrt{p}-1))$, čo predstavuje v porovnaní s prvým prípadom rozdiel $O(\frac{2}{\sqrt{p}})$. To, či tento rozdiel bude zohrávať významnú úlohu určite závisí nielen od veľkosti riešeného problému n, ale aj mnohých ďalších faktorov. Množstvo komunikácie ďalej lineárne narastá s rastúcou vzdialenosťou požadovaných údajov, ktoré je potrebné komunikovať. Napríklad pri výpočtoch prvých a druhých derivácií určitých veličín v niektorom bode mriežky je potrebná informácia len o jednej vrstve najbližších susedných bodov. Inak tomu je pri výpočte vyšších derivácií alebo dokonca Coulombovského potenciálu (1 / vzdialenosť), kde je potrebné komunikovať už celé domény. Preto je dôležité uvedomiť si, že akákoľvek komunikácia medzi paralelne vykonávanými procesmi vyžaduje čas, a tým predlžuje celkový čas vykonávania programu [6].

Funkcionálna dekompozícia

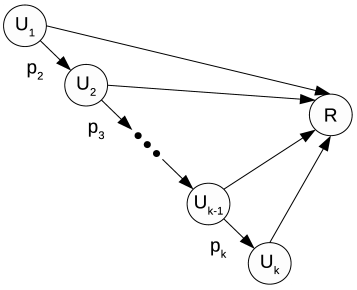
Ako sme už uviedli na začiatku kapitoly, jedná sa o rozdelenie úlohy na niekoľko podúloh tak, že každý procesor vykonáva odlišnú podúlohu a vymieňa si údaje s ostatnými procesormi. Takýto model označujeme ako **MPMD** (Multiple Program Multiple Data). Každú z podúloh je však možné riešiť aj ako samostatný proces v rámci modelu SPMD. Za hlavnú výhodu funkcionálnej dekompozície možno považovať súčasné vykonávanie úloh, ktoré by museli byť inak vykonávané sekvenčne. Na druhej strane je potrebné sa vysporiadať s problémom, keď jednotlivé časti riešenia problému sú rozlične časovo a hardvérovo náročné a ľahko môže dôjsť k nevyváženosti alebo uviaznutiu.

Ako príklad si môžme uviesť testovanie aerodynamických vlastností vozidla. Zatiaľ čo jeden paralelný proces má na starosti simulovania pohybu častíc vzduchu okolo vozidla, druhý paralelný proces môže simulovať reakciu vozidla, ako napríklad prehýbanie sa mäkkých častí z plastu a podobne. Oba tieto procesy si pritom musia navzájom vymieňať informácie, pretože oba skúmané javy sa navzájom ovplyvňujú.

Iným príkladom môže byť algoritmus na sledovanie lúča používaný v počítačovej grafike na renderovanie fotorealistického obrazu z matematicky opísanej scény. Jednou z možností implementácie tohto algoritmu je s použitím modelu **master – worker**, kde jeden hlavný proces rozdeľuje úlohy ostatným procesom, ktoré riešia zadaný podproblém. V tomto algoritme je potrebné pre každý pixel výsledného obrazu vypočítať jeho farbu a to tak, že z pohľadu pozorovateľa je vyslaný lúč, pričom sledujeme, na ktoré objekty lúč narazí, od ktorých sa odrazí a takto získavame hodnotu jeho výslednej farby. Pokiaľ každý z procesov má k dispozícii matematický opis reprezentácie scény, výpočet farby každého z pixelov je nezávislý a možno ho vykonávať bez ohľadu na farbu ostatných pixelov. Z hľadiska efektívnosti nie je vhodné, aby úloha bola rozdeľovaná procesom po jednotlivých pixeloch, ale napríklad po celých riadkoch alebo úsekoch. V okamihu, keď niektorý z procesov dokončí výpočty pre jemu pridelený úsek, požiada hlavný proces o pridelenie ďalšieho ešte nespracovaného úseku zo zoznamu. Viac informácií je možné nájsť v literatúre [7, 8].

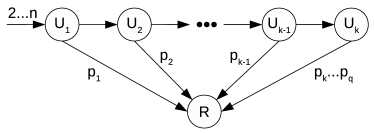
V prípade, že jednotlivé podúlohy na seba nadväzujú, môžme ich znázorniť pomocou acyklického orientovaného grafu, tzv. **grafu závislostí**. Pre pochopenie si zoberme algoritmus Eratostenovo sito na hľadanie prvočísel na intervale $\langle 2, n \rangle$. Princíp algoritmu spočíva vo vypísaní všetkých čísel z uvedeného intervalu a následného postupného vyškrtávania všetkých násobkov už nájdených prvočísel. Ako prvé prvočíslo označíme číslo 2. Následne vyškrtneme zo zoznamu všetky jeho násobky počnúc od jeho druhej mocniny, teda od $2^2 = 4$, pretože tieto čísla už nie sú prvočísla. Ako ďalšie prvočíslo označíme najmenšie nevyškrtnuté číslo 3 a postup opakujeme. Algoritmus končí v momente, keď je nájdené prvočíslo p, pre ktoré platí $p^2 > n$. Tento problém môžme dekomponovať takým spôsobom, že každá z podúloh bude vyškrtávať násobky jej určeného prvočísla. Z algoritmu je zjavné, že jednotlivé podúlohy budú na seba nadväzovať a teda vykonávanie

niektorých podúloh musí predchádzať vykonávaniu ďalších podúloh, ako je znázornené na obrázku 4. Každá z podúloh U_i odstraňuje zo zoznamu násobky svojho prvočísla p_i a nasledujúcej úlohe posiela ďalšie prvočíslo p_{i+1}. Okrem toho odošle toto prvočíslo aj úlohe R, ktorá rozhoduje o tom, kedy sa má algoritmus ukončiť.



Obr. 4: Graf závislostí podúloh

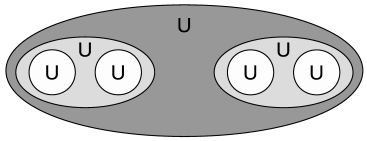
Iné riešenie predošlej úlohy je s využitím **prúdového paralelizmu**. Pri tejto metóde úlohu rozdelíme na k podúloh U a úlohu R, ktorá zbiera výsledky (prvočísla) od ostatných podúloh podľa schémy na obrázku 5. Vstupom úlohy U_1 je celý zoznam čísel z intervalu $\langle 2, n \rangle$ a výstupom je zoznam neobsahujúci násobky čísla 2, pričom podúlohe R je odoslané prvé prvočíslo $p_1 = 2$. Tento zoznam je sekvenčne predávaný nasledujúcej úlohe U_2 a tak postupujeme ďalej až po úlohu U_k, ktorej výsledkom sú už len prvočísla p_k až p_q z daného intervalu.



Obr. 5: Graf závislostí podúloh pre prúdové spracovanie

Pri použití prúdového paralelizmu sa pomerne často môžme stretnúť s problémom, že úlohu nie je možné dekomponovať do dostatočne dlhého radu podúloh, v čoho dôsledku nie je možné využiť na vykonávanie väčšie množstvo procesorov. Ďalšou nevýhodou je postupný nábeh podúloh z dôvodu ich vzájomnej závislosti, preto je výhodné mať čo najväčšiu množinu spracovávaných údajov, aby tento jav bol čo najmenej postrehnuteľný.

Rekurzívna dekompozícia



Obr. 6: Schéma rekurzívnej dekompozície

Rekurzívna dekompozícia predstavuje ďalší z prístupov k riešeniu paralelných úloh. Spravidla je možné ju použiť na úlohy, ktoré je možné riešiť metódou **rozdeľuj a panuj** (divide and conquer) podľa schémy na obrázku 6. Princíp spočíva v delení úlohy na viacero menších navzájom nezávislých podúloh pôvodného problému. Delenie na podúlohy pokračuje rekurzívne až do momentu, kedy je podúloha taká jednoduchá, že má triviálne riešenie. Následne sú výsledky vhodným spôsobom skombinované do finálneho výsledku.

Ako príklad si môžme uviesť triediaci algoritmus quicksort podľa nasledujúceho kódu 1.

Zdrojový kód 1: Algoritmus quicksort

```
1 quicksort(A, l, r)
2 {
3   if l < r then
4     q = vyber pivot a presuň prvky okolo neho
5     quicksort(A, l, q-1)
6     quicksort(A, q+1, r)
7 }
```

Úloha: Navrhnite spôsob implementácie paralelného algoritmu na nájdenie najmenšieho alebo najväčšieho prvku v poli.

Ako môžeme vidieť, pole A[l...r] je v kroku 4 rozdelené na dve časti, prvky menšie ako pivot A[l...q-1] a prvky väčšie alebo rovné ako pivot A[q+1...r]. Tieto dve časti poľa sú následne zatriedené pomocou dvoch úloh na riadkoch 5 a 6 zdrojového kódu.

Pri použití rekurzívnej dekompozície sa výpočtová náročnosť jednotlivých podúloh pri každom kroku znižuje. V dôsledku dynamického vytvárania podúloh môže ľahko nastať situácia, že systém nebude disponovať dostatočným množstvom zdrojov pre vykonávanie tak veľkého množstva podúloh alebo čas réžie vytvárania podúloh je nepomerne väčší ako ušetrený čas paralelizáciou úlohy. Z tohto dôvodu je potrebné rekurziu riadiť dynamicky až počas vykonávania programu a prideliť na vykonávanie väčšiu časť problému (časť poľa na zoradenie) na najnižšej úrovni rekurzie. Na riešenie tejto podúlohy je možné použiť aj iný vhodný nerekurzívny algoritmus.

Exploratívna dekompozícia

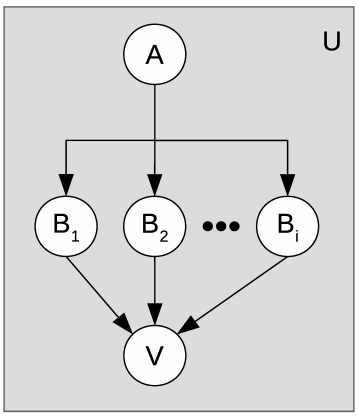
Tento typ dekompozície je vhodný na riešenie úloh, v ktorých je potrebné prehľadávať stavový priestor riešení. Rozdelenie tohto priestoru na niekoľko častí umožňuje ich paralelné prehľadávanie jednotlivými podúlohami. V prípade, že niektorá z podúloh nájde riešenie, odošle o tom informáciu ostatným a vykonávanie programu sa ukončí.

Jednoduchým príkladom je hľadanie správnej cesty v bludisku, pričom pri vstupe do bludiska sa prieskumníci rozdelia, a každý z nich sa vyberie na inú stranu. Úloha je vyriešená v momente, keď niektorý z prieskumníkov objaví východ z bludiska. Množstvo práce pri takejto dekompozícií je priamoúmerné prehľadávanej časti stavového priestoru, a preto môže byť nižšie, ale aj vyššie ako množstvo práce vykonanej porovnateľných sekvenčným programom. Výsledok záleží od pozície výsledku v strome riešení. Túto úlohu je možné formulovať aj ako hľadanie najkratšej alebo najdlhšej cesty. V tomto prípade je potrebné prehľadávanie celého stavového priestoru, a preto je paralelný algoritmus efektívny len vtedy, keď je práca rovnomerne rozdelená medzi jednotlivé podúlohy.

Úloha: Navrhnite spôsob implementácie paralelného algoritmu na hľadanie správneho riešenia v hlavolame Loydová hra "pätnástka".

Špekulatívna dekompozícia

Predstavme si, že potrebujeme získať výsledok vyriešením úlohy `U`, ktorá pozostáva z prvého kroku `A` a následných krokov B_i, podľa schémy na obrázku 7. Pričom od výsledku kroku `A` záleží, ktorý z krokov B_i sa má vykonávať. V prípade, že krok `A` neodosiela žiadne vstupné údaje krokom B_i okrem výberu jednej z podúloh, je možné tieto kroky vykonávať paralelne s použitím viacerých procesorov. Po dokončení prvého kroku `A` je známe, ktorý z krokov B_i má pokračovať a jeho výsledok `V` je použitý ďalej. Výsledky ostatných krokov B_i sa zamietnu. Takémuto riešeniu hovoríme **špekulatívna dekompozícia**, pretože pri začatí vykonávania úloh B_i ešte nie je známe, ktorý z výsledkov sa ďalej použije, a ktoré budú zamietnuté. V dôsledku toho dochádza k vykonávaniu zbytočných výsledkov, ale keďže nie je potrebné čakať na dokončenie kroku `A`, celkový výsledok úlohy `U` získame skôr, v porovnaní so sekvenčným vykonávaním kroku `A` a následne kroku `B`.



Obr. 7: Paralelné vykonávanie podúloh A a B úlohy U

Granularita výpočtu

Poznámka: V niektorej literatúre je možné nájsť aj termín strednozrnná dekompozícia (medium-grained).

Stupeň súbežnosti paralelného programu predstavuje počet súbežne vykonávateľných úloh, na ktoré určitý problém dekomponujeme. Hovoríme o **granularite výpočtu** alebo zrnitosti výpočtu. V prípade, ak máme malý počet zložitejších úloh, ide o **hrubozrnnú** dekompozíciu (coarse-grained). Naopak pokiaľ máme veľké množstvo malých úloh, ide o **jemnozrnnú** dekompozíciu (fine-grained). Granularitu výpočtu je tiež možné definovať ako mieru veľkosti výpočtu a komunikácie [9].

Pre ilustráciu sa môžme vrátiť k príkladu, kde opravovatelia opravujú veľké množstvo testov $n = 1000$, kde každý z testov obsahuje odpovede na $o = 50$ otázok. Ak by sme pri paralelnom riešení tohto problému použili údajovú dekompozíciu tak, že opravenie každého testu bude predstavovať samostatnú úlohu, vznikne nám 1000 úloh, ktoré je možné vykonávať súbežne. Jedná sa o jemnozrnný paralelizmus. Naopak, ak by sme využili model funkcionálnej dekompozície a každý opravovateľ by opravoval jednu z o otázok na všetkých n testoch, vzniklo by nám 50 súbežne riešiteľných úloh a jednalo by sa o hrubozrnný paralelizmus.

Voľba optimálneho stupňa súbežnosti nie je vždy jednoduchá, obzvlášť pokiaľ musíme do úvah zahrnúť aj požiadavky na potrebnú komunikáciu medzi procesmi. Ako príklad nám môže poslúžiť algoritmus na násobenie matice A veľkosti $n \times n$ a vektora X. Výsledkom je vektor $z = Ax$, ktorého prvky sú skalárne produkty vypočítané podľa vzťahu 1, pre i = 1, 2, ..., n.

$$z_i = \sum_{j=1}^{n} a_{i,j} x_j \tag{1}$$

Zo vzťahu je zjavné, že pre získanie výsledku je potrebné vykonať n^2 operácií násobenia a $n(n-1)$ operácia sčítania. Jednotlivé násobenia je možné vykonávať paralelne v n^2 úlohách a následné sčítania komponentov skalárneho produktu je možné vykonať v n úlohách. Takáto dekompozícia poskytuje maximálny stupeň súbežnosti rovný n^2. Za ideálnych okolností by bolo možné takýto algoritmus vykonávať v čase $O(\log n)$ s nákladmi $O(n^2 \log n)$, čo nie je optimálne pri n^2 procesoroch.

Iná možnosť dekompozície spočíva v dekompozícii problému na presne n úloh, kde každá z nich bude počítať hodnotu jedného prvku z_i. Všetkých n úloh je možné vykonávať nezávisle na sebe, a preto bude stupeň súbežnosti rovný n. Za ideálnych okolností by bolo možné výpočty uskutočniť v čase $O(n)$ s použitím n procesorov. Čas výpočtu pre tento model dekompozície je síce vyšší, avšak s optimálnymi nákladmi $O(n^2)$.

V prvom prípade sme dostali vyšší stupeň súbežnosti pri jemnejšiu zrnitosť. Z pohľadu minimalizácie potrebného času na uskutočnenie výpočtu sa zdá, že zrnitosť by mala byť čo najjemnejšia, pretože čas výpočtu krátkych úloh s použitím veľkého počtu procesorov je krátky

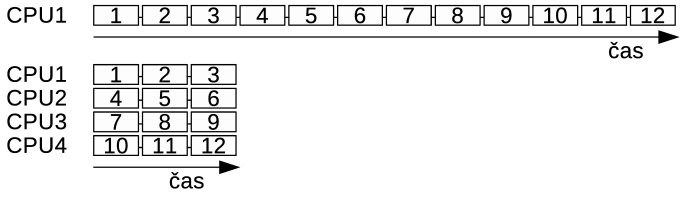
a dosiahneme veľké zrýchlenie vykonávania paralelného programu. Na druhej strane pri vytvorení veľkého počtu malých úloh je možné očakávať vysoké požiadavky na vzájomnú komunikáciu procesorov, ktoré často narastajú priamoúmerne s počtom procesorov, obzvlášť pri použití architektúry s distribuovanou pamäťou. Z tohto dôvodu je potrebné hľadať rozumnú mieru stupňa súbežnosti a granularity pre dosiahnutie optimálneho výsledku [2].

2 Výkonnosť paralelných algoritmov

Hlavným dôvodom vytvárania a spúšťania paralelných programov je zvýšenie efektívnosti využívania výpočtových prostriedkov a skrátenie času, ktorý je potrebné čakať na získanie výsledkov. Aj v tejto oblasti sa uplatňujú určité pravidlá, ktoré je dobré poznať preto, aby sme vedeli efektívne využívať výpočtový výkon a zároveň odhadnúť, aké zrýchlenie môžme od nášho programu očakávať.

Zrýchlenie a efektívnosť paralelného programu

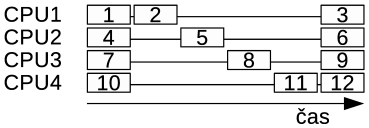
V ideálnom prípade je možné paralelný program vytvoriť rozdelením sériového programu na určitý počet rovnako veľkých podúloh, ktoré je možné riešiť súčasne s použitím p procesorov. Ak sa nám podarí úlohu rozdeliť podľa schémy na obrázku 8 na p podúloh a každy z procesorov bude vykonávať práve jeden proces alebo vlákno riešiace jednu rovnako veľkú podúlohu, pričom nás toto rozdelenie na podúlohy nebude stáť žiaden dodatočný čas, bude náš program vykonaný p-krát rýchlejšie ako pôvodný sériový program. V takomto prípade



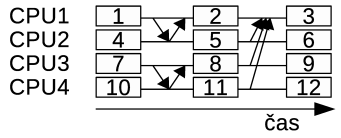
Obr. 8: Rozdelenie úlohy na podúlohy vykonávané sériovo (hore), vykonávané paralelne (dole)

hovoríme o **lineárnom zrýchlení** paralelného programu. Ak označíme čas vykonávania sériového programu ako T_{ser} a čas vykonávania paralelného programu ako T_{par}, tak potom platí, že $T_{par} = T_{ser}/p$.

V reálnej praxi však takýto prípad nastáva veľmi zriedka, pretože s rozdelením úlohy a vytváraním procesov alebo vlákien je spojená istá réžia trvajúca určitý čas (overhead). Pri použití modelu so zdieľanou pamäťou v určitých momentoch počas vykonávania programu nastane situácia, keď sa viaceré alebo aj všetky vlákna stretnú v kritickej oblasti (napríklad prístup k zdieľanej premennej), do ktorej je potrebné riadiť prístup. Z tohto dôvodu je možné, aby do kritickej oblasti mohlo vstúpiť v tom istom čase iba jedno vlákno tak, ako je znázornené na obrázku 9, v čoho dôsledku sa táto časť programu serializuje, čo predlžuje čas vykonávania paralelného programu.



Obr. 9: Vykonávanie kritickej oblasti viacerými vláknami



Obr. 10: Komunikácia po sieti medzi procesmi

Pri použití modelu s distribuovanou pamäťou zas nastáva situácia, že procesy si potrebujú medzi sebou vymieňať určité údaje tak, ako je znázornené na obrázku 10. Táto komunikácia sa odohráva pomocou počítačovej siete, ktorá je rádovo pomalšia než prístup k údajom uloženým v pamäti počítača.

Mierou, ktorá porovnáva rýchlosť vykonávania sériového a paralelného programu je **zrýchlenie paralelného programu** S, ktoré definujeme vo vzťahu 2.

$$S = \frac{T_{ser}}{T_{par}} \tag{2}$$

Pri lineárnom zrýchlení paralelného programu platí, že $S = p$, čo je v praxi málo pravdepodobné. Dokonca, čím viac zvyšujeme počet procesorov, tým viac sa zrýchlenie vzdiaľuje od lineárneho. Tento jav môžme pochopiť aj intuitívne, pretože čím viac vlákien bude vstupovať do kritickej oblasti, tým dlhšie bude trvať, kým sa do nej dostatnú všetky vlákna. Podobne aj výmena údajov pri komunikácií po sieti bude trvať dlhšie, pretože bude nutná komunikácia medzi viacerými procesmi. Pomer medzi zrýchlením paralelného programu a počtom použitých procesorov označujeme ako **efektívnosť paralelného programu** a platí vzťah 3.

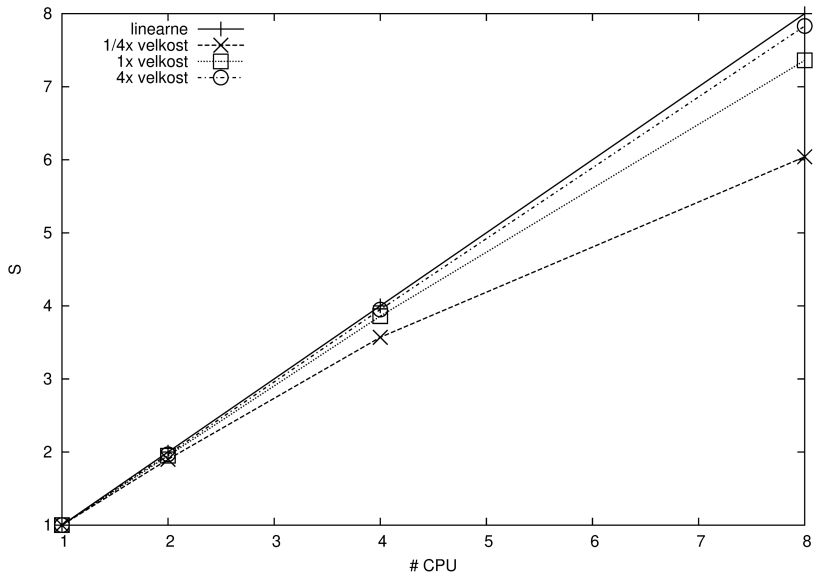
$$E = \frac{S}{p} = \frac{T_{ser}}{p \cdot T_{par}} \tag{3}$$

Na čas vykonávania paralelného programu, jeho zrýchlenie a efektívnosť má okrem počtu procesorov významný vplyv aj veľkosť samotnej riešenej úlohy. V tabuľke 1 sú uvedené výsledky týchto vlastností pre paralelný program numerického výpočtu hodnoty integrálu vzhľadom na počet procesorov a veľkosť riešenej úlohy n. Hodnoty sú uvedené pre 1 až 8 procesorov a päť veľkostí úlohy (štvrtinová, polovičná, pôvodná, dvojnásobná a štvornásobná veľkosť).

Tabuľka 1: Zrýchlenie a efektívnosť paralelného programu (S – zrýchlenie, E – efektívnosť)

Počet procesorov	p	1	2	4	8
Veľkosť úlohy					
$n/4$	S	1	1,90	3,57	6,04
	E	1	0,95	0,89	0,76
$n/2$	S	1	1,94	3,76	6,89
	E	1	0,97	0,94	0,86
n	S	1	1,95	3,86	7,36
	E	1	0,98	0,97	0.92
$2n$	S	1	1,97	3,92	7,67
	E	1	0,98	0,98	0,96
$4n$	S	1	1,97	3,95	7,83
	E	1	0,98	0,99	0,98

Z tabuľky je jasne vidieť, že zrýchlenie paralelného programu S je vo všetkých prípadoch nižšie ako počet procesorov p, avšak so zväčšujúcou sa veľkosťou riešenej úlohy n sa čoraz viac približuje k hodnote p (viď. obrázok 11).



Obr. 11: Zrýchlenie paralelného programu pre rôzne veľkosti problému

Tiež si môžme všimnúť, že efektívnosť paralelného programu riešiacich tú istú úlohu klesá so zvyšujúcim sa počtom procesorov, naopak s rastúcou veľkosťou problému sa efektivita pre rovnaký počet procesorov zvyšuje (viď. obrázok 12).

Toto správanie je možné zdôvodniť potrebou času na **réžiu paralelizácie**. Preto do našich úvah zahrnieme aj čas réžie T_{rez}, a teda platí vzťah 4.

$$T_{par} = \frac{T_{ser}}{p} + T_{rez} \tag{4}$$

So zväčšujúcou sa veľkosťou problému zväčša čas réžie T_{rez} narastá pomalšie ako čas vykonávania sériového programu T_{ser}. V takomto prípade bude zrýchlenie a efektívnosť paralelného programu narastať.

V bežných paralelných programoch môžu nastať tieto okolnosti, ktoré vedú k predĺženiu času vykonávania programu:

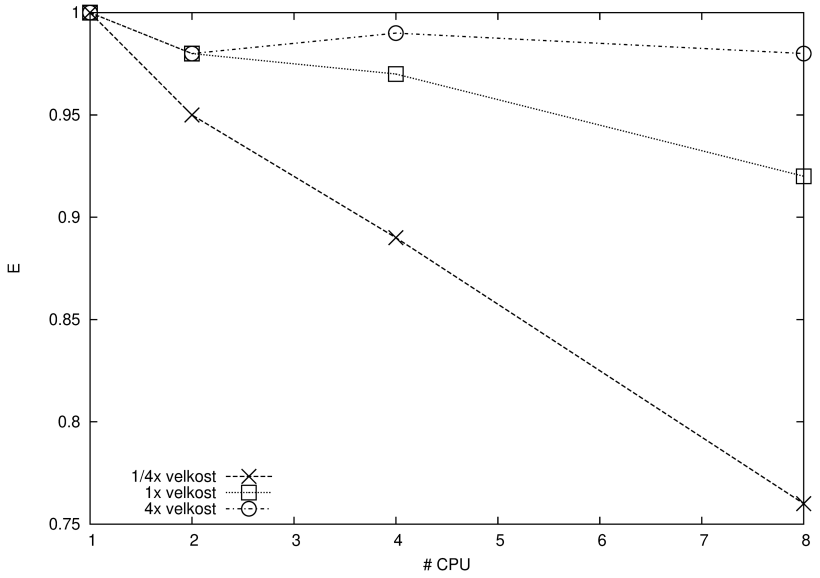
- **obmedzenia algoritmu** – niektoré časti programu na seba nadväzujú,
- **úzke miesto (bottleneck)** – v programe sa môže pracovať so zdieľanými prostriedkami,
- **réžia inicializácie** – aktivácia ďalších procesov alebo vlákien paralelného programu,
- **komunikácia** – výmena údajov medzi jednotlivými podúlohami,
- **nevyváženosť** – jednotlivé podúlohy nie sú rovnako veľké a ich vykonanie trvá rôzne dlhý čas.

Otázne však je, akú hodnotu by sme mali považovať za správnu pre T_{ser}. Podľa niektorých autorov by sa mala brať do úvahy hodnota

Úloha: Majme sériový program, ktorý je vykonateľný v čase $T_{ser} = n^2$ mikrosekúnd. Čas vykonávania paralelného programu je $T_{par} = \frac{n^2}{p} + log_2(p)$. Napíšte program, ktorý zistí zrýchlenie a efektívnosť tohto paralelného programu pre rôzne kombinácie n a p. Spustite Váš program pre $n = 10, 20, 40, ..., 320$ a $p = 1, 2, 4, ..., 128$. Ako sa bude správať zrýchlenie a efektívnosť pokiaľ budeme meniť len jednu z hodnôt a druhú necháme nezmenenú?

Úloha: Predpokladajme, že $T_{par} = \frac{T_{ser}}{p} + T_{rez}$ a počet procesorov p sa nebude meniť.

a) Ukážte, že ak T_{rez} narastá pomalšie ako T_{ser}, tak efektívnosť paralelného programu bude narastať s rastúcou veľkosťou problému.
b) Ukážte, že ak T_{rez} narastá rýchlejšie ako T_{ser}, tak sa efektívnosť paralelného programu bude znižovať s rastúcou veľkosťou problému.



Obr. 12: Efektívnosť paralelného programu pre rôzne veľkosti problému

času vykonávania najrýchlejšieho sériového programu na najrýchlejšom dostupnom procesore. V praxi sa však často uvažuje s hodnotou získanou vykonávaním paralelného programu s použitím iba jedného procesora paralelného systému [10].

Amdahlovo pravidlo

Ako sme sa už zmienili, pri písaní paralelných programov zvyčajne narazíme na časti programu, ktoré nie je možné paralelizovať a je potrebné ich vykonať sériovo. V 60-tých rokoch 20. storočia Gene Amdahl vyslovil empirické pravidlo, ktoré hovorí o tom, že možné zrýchlenie paralelného programu vzhľadom k počtu procesorov je významne limitované, pokiaľ sa v programe vyskytujú časti, ktoré nie je možné paralelizovať. Pre toto pravidlo sa zaužíval pojem **Amdahlovo pravidlo** [11].

Predstavme si, že máme sériový program, pričom tento vieme z 90 % paralelizovať a zvyšných 10 % musíme vykonať sériovo. Ďalej uvažujme, že efektívnosť paralelizácie tých 90 % programu je rovná $E = 1$, a teda zrýchlenie paralelnej časti programu je rovné počtu procesorov $S = p$. Za predpokladu, že $T_{ser} = 20$ sekúnd, tak čas vykonávania paralelizovateľnej časti bude $\frac{0,9T_{ser}}{p} = \frac{18}{p}$. Čas vykonávania sériovej časti bude $0,1T_{ser} = 2$. Potom celkový čas behu paralelného programu môžme vyjadriť vzťahom 5 a zrýchlenie paralelného programu vzťahom 6, kde α predstavuje podiel paralelizovateľnej časti programu.

$$T_{par} = \frac{\alpha T_{ser}}{p} + (1-\alpha)T_{ser} = \frac{0,9 \cdot 20}{p} + 0,1 \cdot 20 = \frac{18}{p} + 2 \tag{5}$$

$$S = \frac{T_{ser}}{\frac{\alpha T_{ser}}{p} + (1-\alpha)T_{ser}} = \frac{20}{\frac{0,9 \cdot 20}{p} + 0,1 \cdot 20} = \frac{20}{\frac{18}{p} + 2} \tag{6}$$

V prípade, že by sme neustále zväčšovali počet procesorov p, časť výrazu obsahujúca p v menovateli by sa čoraz viac blížila k nule a celkový čas paralelného programu by nemohol byť menší než $0,1T_{ser}$, v dôsledku čoho menovateľ vo vzťahu 6 nemôže byť menší ako $0,1T_{ser}$. Preto pre výslednú hodnotu dosiahnutého zrýchlenia paralelného programu platí vzťah 7.

$$S \leq \frac{T_{ser}}{(1-\alpha)T_{ser}} = \frac{20}{0,1 \cdot 20} = \frac{20}{2} = 10 \tag{7}$$

Na základe Amdahlovho pravidla teda platí, že v prípade ak vieme efektívne paralelizovať 90 % programu a na vykonávanie použijeme hoci aj veľké množstvo procesorov, zrýchlenie paralelného programu nebude lepšie ako 10. Vo všeobecnosti teda platí, že zrýchlenie paralelného programu nemôže byť vyššie ako $\frac{1}{1-\alpha}$.

Toto pravidlo nám poskytuje odpoveď na otázku, do akej miery je možné zrýchliť vykonávanie sériového programu riešiaci problém určitej veľkosti. Z pohľadu používateľa by sme pri vykonávaní paralelného programu mali použiť čo najväčšie možné množstvo procesorov, pretože tak je možné čo najviac minimalizovať čas jeho vykonávania. Avšak Amdahlovo pravidlo nezahŕňa v sebe niektoré dôležité skutočnosti a to, že s rastúcou veľkosťou problému sa zvyšuje pomer v prospech paralelne vykonávateľnej časti na úkor sériovo vykonávanej časti. O tejto problematike podrobnejšie pojednáva **Gustafsonovo pravidlo** [12]. Zároveň pokiaľ nás k paralelizácií programu priviedla potreba väčšieho množstva pamäte, je možné akceptovať aj riešenie, kde sa použitie veľkého množstva procesorov môže javiť ako mrhanie výpočtovými zdrojmi.

Úloha: Paralelný program, ktorého zrýchlenie je vyššie ako počet procesorov p označujeme ako **superlineárne zrýchlenie**. Toho je možné dosiahnuť prekonaním limitov zariadení. Napríklad sériový program musí používať pri výpočtoch pre uchovávanie údajov pomalšie sekundárne úložisko, zatiaľ čo paralelný program, s použitím modelu s distribuovanou pamäťou, je schopný všetky údaje uchovať v hlavnej pamäti. Uveďte ďalšie príklady, ktoré umožňujú prekonať limity zariadení a dosiahnuť väčšie zrýchlenie ako p.

3 Škálovateľnosť

Pojem škálovateľnosť sa používa vo viacerých významoch od hardvérových architektúr až po škálovateľnosť paralelného programu. Vo všeobecnosti je možné tento pojem chápať, ako schopnosť použitej technológie pre riešenie problémov s čoraz väčšou veľkosťou. V našom ponímaní budeme pojem **škálovateľnosť** vnímať z pohľadu paralelného programu. Predstavme si, že máme paralelný program, ktorý je schopný riešiť určitý problém s danou veľkosťou vstupu n na určitom počte procesorov p, pričom je vykonávaný s určitou efektívnosťou E. Za predpokladu, že zvýšime počet procesorov, ktoré budú riešiť úlohu s rovnakou efektívnosťou hovoríme, že paralelný program je škálovateľný.

Predpokladajme, že čas vykonávania sériového programu udávaný v milisekundách $T_{ser} = n$, a teda je rovný veľkosti vstupu riešenej úlohy. Tiež predpokladajme, že $T_{par} = \frac{n}{p} + 1$. Potom platí vzťah 8.

$$E = \frac{n}{p(\frac{n}{p}+1)} = \frac{n}{n+p} \tag{8}$$

Úloha: Na vyriešenie problému programom je potrebné celkom vykonať 10^{12} inštrukcií. Predpokladajme, že jednojadrový procesor dokáže tento program dokončiť za 10^6 sekúnd (asi 11,6 dňa). Z toho vyplýva, že procesor je schopný vykonať približne 10^6 inštrukcií za sekundu. Uvedený program spararelizujeme pre systém s distribuovanou pamäťou tak, že ho bude možné spustiť na p procesoroch. Každý procesor bude teda musieť vykonať $\frac{10^{12}}{p}$ inštrukcií a odoslať $10^9(p-1)$ správ. Predpokladajme, že paralelný program nie je zaťažený žiadnou ďalšou réžiou paralelizácie. Za dokončenie programu považujeme stav, keď každý z procesorov vykonal všetky inštrukcie, ktoré mu boli pridelené a odoslal všetky správy. Vypočítajte, ako dlho bude trvať vyriešenie úlohy s použitím 1000 procesorov, pričom každý z procesorov je rovnako rýchly ako jednojadrový procesor. Odoslanie jednej správy bude trvať:

a) 10^{-9} sekundy,
b) 10^{-3} sekundy.

Úloha: Je paralelný program, ktorý dosahuje lineárne zrýchlenie silne škálujúci? Zdôvodnite Vašu odpoveď.

Ak pre riešenie takejto úlohy použijeme k-násobne väčší počet procesorov, zo vzťahu 8 vyplýva, že ak chceme zachovať rovnakú efektívnosť paralelného programu, musíme primerane – x-násobne zväčšiť aj veľkosť vstupu. Preto označme takto zvýšený počet procesorov kp a zväčšenú veľkosť vstupu xn, pričom musí platiť vzťah 9.

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp} \tag{9}$$

Z uvedeného nám vyplýva, že pre $x = k$ je možné vzťah 9 upraviť nasledovne:

$$\frac{xn}{xn+kp} = \frac{kn}{kn+kp} = \frac{n}{n+p} \tag{10}$$

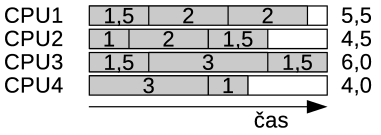
V podstate to znamená, že paralelný algoritmus je **škálovateľný** vtedy, ak vieme zvýšiť veľkosť vstupu úlohy v rovnakej miere ako počet procesorov, ktoré úlohu riešia pri zachovaní rovnakej efektívnosti paralelného programu [10].

V niektorej literatúre sa môžme stretnúť aj s pojmom **silne škálujúci** paralelný program. Hovoríme o ňom vtedy, ak môžeme zvýšiť počet procesorov riešiacich úlohu bez toho, aby sa zvýšilo množstvo práce (výpočtov, komunikácie, atď.), ktorú je potrebné vykonať. To znamená, že zachováme približne rovnakú efektívnosť paralelného programu bez potreby zväčšenia veľkosti vstupu úlohy. V takomto prípade je možné za hlavný cieľ paralelizácie považovať skrátenie času riešenia danej úlohy. Naopak pokiaľ hlavným cieľom paralelizácie nie je skrátenie času riešenia úlohy, ale riešenie väčšieho problému (limitovaného veľkosťou pamäte), je vhodné, aby veľkosť problému škálovala s ľubovoľnou kladnou mocninou počtu použitých procesorov. V takomto prípade množstvo vykonanej práce škáluje tiež s touto mocninou počtu procesorov. Pojem **slabo škálujúci** paralelný program sa používa obzvlášť vtedy, keď vieme zachovať približne rovnakú efektívnosť paralelného programu tak, že budeme zväčšovať veľkosť vstupu úlohy v rovnakej miere, ako počet procesorov riešiacich túto úlohu [6, 10].

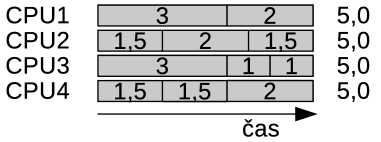
4 Vyvažovanie záťaže

Jedným z najvýznamnejších dôvodov pre vytváranie a používanie paralelných programov na riešenie výpočtovo náročných úloh je skrátenie času ich výpočtu. V momente, keď už máme zvolený vhodný model dekompozície úlohy na podúlohy, vieme, čo budú jednotlivé podúlohy robiť, a ako to budú vykonávať, môžme tieto priradiť na vykonávanie jednotlivým procesorom. Našou snahou je manažovať vykonávanie týchto podúloh tak, aby všetky procesory boli zaneprázdnené čo najrovnomernejšie a celkový čas riešenia problému bol čo najkratší. Toho je možné docieliť tým, že budeme minimalizovať čas, keď niektoré procesory nemajú žiadnu prácu a musia čakať, kým svoju prácu dokončia ostatné procesory. Tomuto procesu hovoríme

vyvažovanie záťaže. Na obrázkoch 13 a 14 sú znázornené jednotlivé podúlohy v podobe obdĺžnikov s číslom, ktoré predstavuje dĺžku trvania vykonávania podúlohy v časových jednotkách. V prvom prípade môžeme vidieť, že podúlohy neboli pridelené na vykonávanie procesorom úplne optimálne, takže niektoré procesory museli po skončení vykonávania svojich podúloh čakať na dokončenie ostatných, a teda celkový čas dokončenia bol 6 časových jednotiek. V druhom prípade sa podúlohy podarilo prideliť procesorom tak, že všetky procesory dokončili im pridelené podúlohy v rovnakom čase, a teda celkový čas dokončenia bol 5 časových jednotiek.



Obr. 13: Nevyvážené rozvrhovanie podúloh



Obr. 14: Vyvážené rozvrhovanie podúloh

Je potrebné uvedomiť si, že nevyvážené vykonávanie podúloh vedie okrem predĺženého času dokončenia úlohy aj k celkovému zvýšeniu nákladov. Tieto môžme jednoducho vyjadriť ako súčin času vykonávania úlohy a počtu použitých procesorov $T \cdot p$. V prvom nevyváženom prípade rozvrhovania podľa obrázku 13 by celkové náklady predstavovali $6 \cdot 4 = 24$. V druhom vyváženom prípade rozvrhovania podľa obrázku 14 by celkové náklady predstavovali $5 \cdot 4 = 20$, čo predstavuje takmer pätinovú úsporu nákladov.

Za predpokladu, že označíme počet použitých procesorov p a časy dokončenia úloh pre jednotlivé procesory T_i, kde $i = 1, 2, 3, ..., p$, čas dokončenia úlohy posledným procesorom $T_{max} = max(T_i)$, môžme definovať **mieru nevyváženosti** podľa vzťahu 11. Táto v princípe vyjadruje pomer času, kedy procesory nevykonávajú žiadnu podúlohu a musia čakať k celkovému času dokončenia všetkých podúloh.

$$L = \frac{\sum_{i=1}^{p} T_{max} - T_i}{pT_{max}} \tag{11}$$

Pri riešení reálnych úloh nie je dosiahnutie optimálneho rozvrhovania vôbec jednoduché. V prvom rade je vôbec náročné určiť alebo odhadnúť potrebný čas na vykonanie jednotlivých podúloh. Druhým dôvodom môže byť heterogenita výpočtových prostriedkov, ako napríklad rýchlosť CPU alebo použité softvérové nástroje. V neposlednom rade je problematické odhadnúť dopad komunikácie medzi podúlohami na rýchlosť ich vykonávania. Okrem toho je potrebné brať do úvahy aj samotný operačný systém, jeho služby a zdržanie kvôli systémovým volaniam. Z uvedených dôvodov nie je možné vyvažovanie záťaže manažovať za každých okolností optimálne, ale je možné sa čo najviac k optimálnemu riešeniu priblížiť.

Statické vyvažovanie záťaže

Statické vyvažovanie záťaže je špecifické tým, že rozdelenie úloh je známe skôr, než ich začnú jednotlivé procesory vykonávať. Pre vytvorenie čo najlepšieho rozvrhu je potrebné zobrať v úvahu viacero aspektov, ako napríklad veľkosť podúloh z pohľadu počtu vykonaných operácií, množstvo spracovávaných údajov alebo frekvenciu a množstvo potrebnej komunikácie medzi podúlohami. V praxi je však veľmi obtiažne, ba dokonca častokrát vopred vôbec nemožné tieto

Poznámka: Nie je známy postup riešenia NP-úplného problému v polynomiálnom čase, to znamená, že čas na jeho riešenia rastie asymptoticky rýchlejšie ako polynomiálne (zväčša exponenciálne). V dôsledku toho už riešenie stredne veľkých NP-úplných problémov s použitím dostupnej výpočtovej techniky môže trvať roky.

parametre podúloh kvantifikovať, a aj keby sa nám to podarilo, zostavenie optimálneho rozvrhu pre pridelenie jednotlivých podúloh procesorom je NP-úplný problém.

Z týchto dôvodov sa pristupuje k použitiu jednoduchších a rýchlejších rozvrhovacích algoritmov alebo algoritmus s prvkami heuristiky. Jedným z najčastejšie používaných algoritmov je **algoritmus round-robin**. Jeho princíp je v postupnom prideľovaní podúloh procesorom v poradí. Po pridelení podúlohy poslednému procesoru sa opäť pokračuje v prideľovaní ďalších podúloh od prvého procesora, až kým nie sú všetky podúlohy priradené procesorom. Ďalšou z jednoduchých možností je použitie **algoritmu náhodného prideľovania** podúloh, ktorý prideľuje úlohy procesorom v náhodnom poradí. Sofistikovanejšie spôsoby statického vyrovnávania záťaže spočívajú v použití genetických algoritmov alebo simulovaného žíhania a podobne.

Pri písaní programov využívajúcich statické vyvažovanie záťaže je nevyhnutné poznať samotný model rozvrhovania úloh už vo fáze návrhu paralelného programu. Na základe použitého modelu dekompozície problému je známe, čo bude úlohou jednotlivých podúloh, a teda je možné tieto podúlohy navrhnúť tak, aby po pridelení jednotlivým procesorom bolo ich vykonávanie čo najvyváženejšie. Spravidla sa používa najmä pri údajovej alebo funkcionálnej dekompozícií úloh.

Pri údajovej dekompozícií rozdelíme určité množstvo údajov na rovnako alebo porovnateľne veľké časti. Tieto časti sú potom prideľované procesorom na spracovanie, pričom sa očakáva, že spracovanie jednej časti pridelenej jednému procesoru bude trvať približne rovnako dlho, ako pridelenie druhej časti druhému procesoru. Vďaka tomu je možné rovnomerne rozložiť záťaž medzi všetky použité procesory. Rozdelenie údajov na časti býva zväčša jednoduché pokiaľ pracujeme s pravidelnými údajovými štruktúrami, ako sú napríklad vektory alebo n-rozmerné matice. Obtiažnejšie je to už pri použití nepravidelných údajových štruktúr. Príkladom môžu byť výpočty spojené s meteorologickými podmienkami na určitom geografickom území. V takomto prípade je vhodné navzorkovať skúmané územie s jemným rastrom a porovnať podmienky pre jednotlivé segmenty rastra. Následne segmenty s podobnými vlastnosťami spojiť do väčších celkov a tie prideliť ako podúlohy procesorom. Medzi ďalšie z dôležitých a často sa vyskytujúcich, avšak problematických údajových štruktúr, patria grafy. Riešením je rozdeliť graf na k podmnožín, pričom k je rovnaké ako počet procesorov a každú z nich prideliť na vykonávanie jednému z procesorov. Tiež je žiadúce, aby jednotlivé podmnožiny mali medzi sebou čo najmenej hrán.

Pri spracovaní veľkého množstva údajov je takéto vyvažovanie záťaže postačujúce. Stačí pokiaľ údaje rozdelíme na rovnako veľké podmnožiny a staticky ich priradíme na spracovanie procesorom. Charakteristickou črtou úloh, ktoré je možné riešiť pomocou údajovej dekompozície je, že stupeň súbežnosti paralelného programu zvyčajne rastie s rastúcou veľkosťou problému. V dôsledku toho je možné efektívne riešiť veľké úlohy pomocou veľkého počtu procesorov.

Pri funkcionálnej dekompozícií rozdelíme úlohu na rôzne časti, pričom

jednotlivé časti sú vykonávané ako samostatné funkcie. Tieto môžu častokrát na seba nadväzovať, čo môžeme opísať pomocou acyklického grafu závislosti, kde vrcholy zodpovedajú podúlohám a hrany ich závislosti. Na základe neho je možné určiť potrebu komunikácie, synchronizácie, ale aj samotného poradia pre vykonávanie jednotlivých podúloh. Podstata funkcionálnej dekompozície spočíva v rozdelení grafu závislostí na podmnožiny tak, aby čas riešenia problému bol čo najkratší. Dvojicu kritických prípadov predstavujú prípady, keď sú všetky podúlohy vykonávané jedným procesorom, čo eliminuje potrebu komunikácie medzi podúlohami za cenu sériového vykonávania programu. Druhým extrémom je vykonávanie každej z podúloh iným procesorom, čo síce poskytuje maximálnu mieru paralelizmu, ale aj komunikácie. Pre hľadanie optimálneho riešenia je potrebné zvoliť rozumnú mieru stupňa súbežnosti paralelného programu a mieru potrebnej komunikácie medzi riešenými podúlohami.

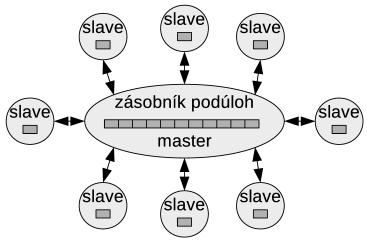
Použitie takejto metódy je efektívne v prípade, že stupeň súbežnosti paralelného programu (počet nezávisle paralelne riešiteľných podúloh) je dostatočne vysoký. Priradenie podúloh procesorom sa vykonáva staticky na základe grafu závislostí, pričom je potrebné čo najviac minimalizovať množstvo potrebnej komunikácie a interakcií medzi podúlohami. Druhou z možností je použiť dynamické prideľovanie podúloh, kde jeden z procesov bude rozdeľovať podúlohy procesorom, pričom bude zohľadňovať prípadnú komunikáciu medzi nimi [2].

Dynamické vyvažovanie záťaže

Na rozdiel od statického vyvažovania záťaže pri dynamickom vyvažovaní záťaže sú podúlohy prideľované na vykonávanie procesorom až počas vykonávania paralelného programu. V prípadoch, keď sú podúlohy generované až za behu, alebo ak nie je možné ani len odhadnúť čas vykonávania podúloh, je použitie dynamického vyvažovania jedinou možnosťou. V takomto prípade je procesoru pridelená podúloha až v momente, keď dokončil predchádzajúcu, v čoho dôsledku sa skracuje čas nečinnosti procesorov pri čakaní. Dynamické vyvažovanie záťaže môže viesť aj k zhoršeniu efektívnosti, napríklad v prípade keď potrebujeme spracovať veľké množstvo údajov, avšak vykonávaná operácia s nimi nie je príliš výpočtovo zložitá, a teda čas získaný dynamickým pridelením úlohy inému procesoru nemusí byť dostatočný v porovnaní s časom potrebným na prenos údajov po sieti.

Metódy dynamického vyvažovania záťaže môžeme rozdeliť na:

- centralizované,
- decentralizované,
- distribuované.



Obr. 15: Centralizovaná metóda dynamického vyvažovania záťaže

Centralizovaná metóda

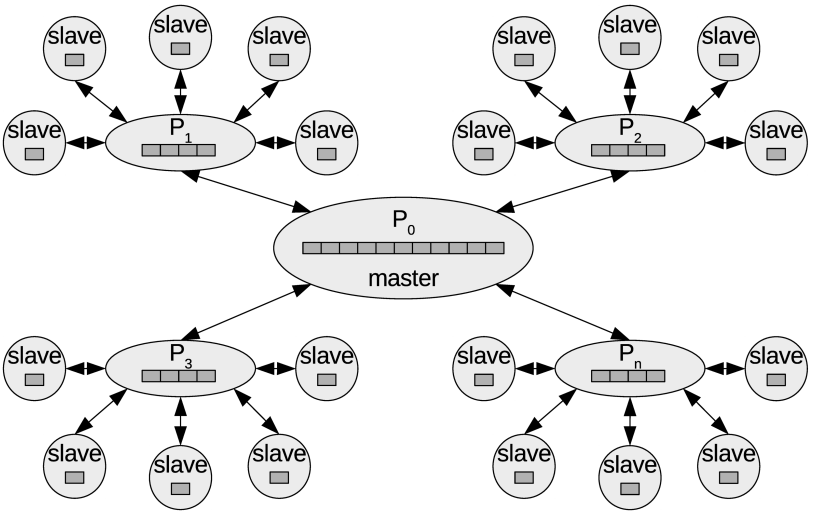
Pri centralizovanej metóde dynamického vyvažovania záťaže je zoznam jednotlivých podúloh uložený v centralizovanom zásobníku, ku ktorému môžu pristupovať všetky procesory. Zásobník úloh môže byť uložený buď v zdieľanej pamäti alebo v jednom **hlavnom (master) procese**, ktorý prideľuje úlohu ostatným **podriadeným (slave) procesom** a zbiera požiadavky na pridelenie ďalšej podúlohy. Schématické znázornenie tejto metódy môžeme vidieť na obrázku 15.

Keď niektorý z podriadených procesov dokončí svoju podúlohu, požiada hlavný proces o pridelenie ďalšej. Ak počas vykonávania vznikne potreba vytvoriť ďalšiu novú podúlohu, táto je hlavným procesom zaradená do zoznamu. Ak majú všetky podúlohy približne rovnakú veľkosť a charakter stačí, ak použijeme jednoduchý rád **FIFO**. V prípade, že veľkosť úloh je rozdielna, je výhodnejšie najprv vykonávať väčšie úlohy a až potom menšie. V takomto prípade je možné použiť na implementáciu zoznamu **prioritný rad**.

Centralizovaná metóda dynamického vyvažovania záťaže je pomerne jednoduchá a využíva sa najmä pokiaľ pracujeme s menším počtom procesorov a využíva sa hrubozrnný paralelizmus. V opačnom prípade by mohlo prísť k zahlteniu hlavného procesu požiadavkami od ostatných procesov.

Decentralizovaná metóda

Na odstránenie problému zahltenia hlavného procesu pri použití veľkého počtu procesorov je možné rozdeliť zásobník s podúlohami na niekoľko menších podzásobníkov podľa schémy na obrázku 16. Na začiatku hlavný proces P_0 rozdelí podúlohy procesom P_1 až P_n a tie následne prideľujú podúlohy svojim podriadeným procesom v rámci skupiny.

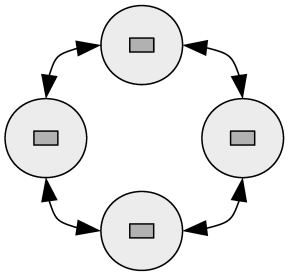


Obr. 16: Decentralizovaná metóda dynamického vyvažovania záťaže

Distribuovaná metóda

Pri použití distribuovanej metódy dynamického vyvažovania záťaže procesy nie sú rozdelené na hlavný a podriadené, ale každý z procesov si manažuje svoj vlastný zoznam podúloh a zároveň ich aj vykonáva. Vyvažovanie záťaže sa uskutočňuje formou **migrovania procesov** medzi procesormi. Tieto môžu migrovať buď na základe požiadavky niektorého z procesorov, ktorý má prázdny alebo takmer prázdny zoznam svojich úloh. Z praktických skúseností sa tento model veľmi neosvedčuje vo viac zaťažených systémoch. Druhá možnosť inicializácie migrácie procesu môže nastať zo strany procesora, ktorý je preťažený a odošle úlohy tým procesorom, ktoré sú ešte schopné proces akceptovať. Takýto model je výhodný pri menej zaťažených systémoch. Jednoduchá schéma pre tento model je znázornená na obrázku 17, avšak je ho možné aplikovať aj v iných topológiach, ako napríklad n-rozmerná hyperkocka.

S použitím dynamického vyvažovania záťaže sa obzvlášť pri použití jemnozrnného paralelizmu spája vyššia požiadavka na réžiu paralelizácie v porovnaní so statickým vyvažovaním záťaže. Ďalším z problémov distribuovanej metódy dynamického vyvažovania záťaže je problém detekcie ukončenia distribuovaného výpočtu. Za predpokladu, že vieme zadefinovať určitú podmienku, ktorá musí byť splnená aspoň jedným z procesov, môže tento proces o jej splnení informovať poslaním správy všetky ostatné procesy, ktoré sa po prijatí správy môžu ukončiť. Komplikovanejší prípad je, keď podmienka pre ukončenie distribuovaného výpočtu musí byť splnená globálne vo všetkých procesoch. V takom prípade je nevyhnutná ďalšia komunikácia medzi procesmi, pretože každý z procesov má iba lokálnu informáciu o splnení podmienky.



Obr. 17: Distribuovaná metóda dynamického vyvažovania záťaže

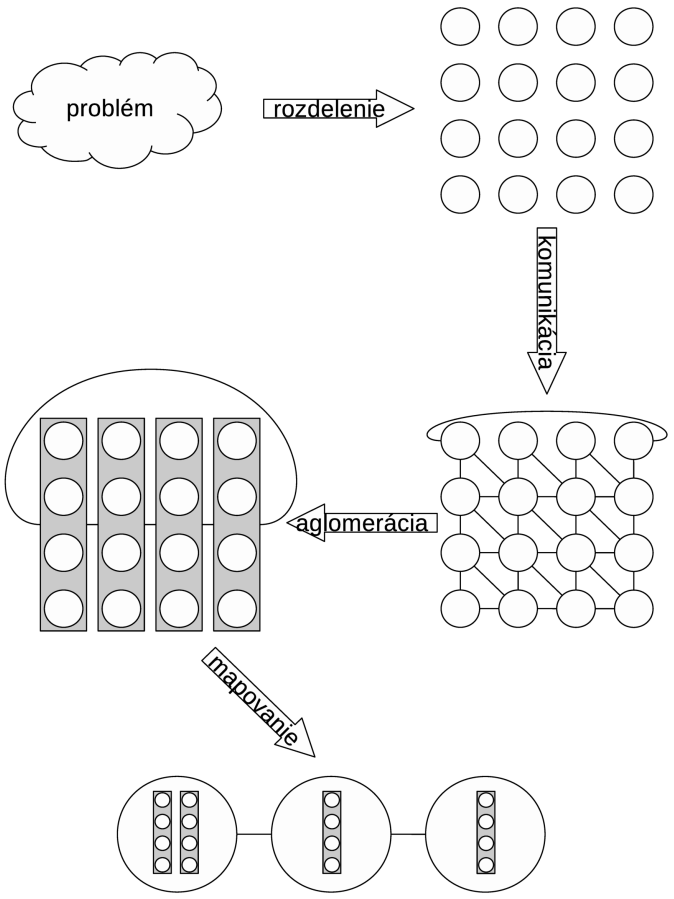
5 Návrh paralelného programu

Samotnému návrhu paralelného programu takmer stále predchádza existencia určitej implementácie sériového programu, z ktorého návrhu vychádzame. To je na jednej strane dobré, pretože nie je potrebné vymýšľať nový program. Avšak na druhej strane neexistuje univerzálny alebo automatický postup, ako existujúci sériový program pretransformovať na efektívne pracujúci paralelný program. Dôvody, ktoré nás priviedli k potrebe paralelného programu zväčša bývajú dva. Za prvé potrebujeme v rozumnom čase zrealizovať časovo náročné výpočty alebo potrebujeme spracovať obrovské množstvo údajov. V každom prípade vieme, že danú výpočtovo náročnú úlohu potrebujeme rozdeliť medzi jednotlivé procesy, prípadne vlákna tak, aby každý z procesov dostal na spracovanie približne rovnako veľa práce. Okrem toho sa takmer vždy nezaobídeme bez synchronizácie a komunikácie medzi procesmi. Hoci odoslanie správy druhému procesu môže byť vykonané asynchrónne, prijímanie správy je synchrónne, a preto pokiaľ správa ešte nebola odoslaná prvým procesom, druhý proces bude zablokovaný až do momentu doručenia správy od prvého procesu.

Napriek uvedeným skutočnostiam existuje všeobecne funkčný postup, aplikovaním ktorého je možné úspešne navrhnúť paralelnú implementáciu paralelného programu. Ian Foster navrhol vo svojej práci [3] postup z niekoľkých krokov:

1. **Rozdelenie** alebo dekompozícia – rozdelenie úlohy a údajov na malé časti, pričom berieme ohľad na to, aby mohli byť tieto časti spracovávané paralelne.
2. **Komunikácia** – navrhnutie nevyhnutnej komunikácie medzi časťami z predchádzajúceho kroku potrebných pre vykonanie celej úlohy.
3. **Aglomerácia** alebo agregácia – spojenie úloh, ktoré na seba nadväzujú alebo vykonávajú podobnú prácu do väčších celkov, čím sa zníži celkový počet podúloh a množstvo potrebnej komunikácie medzi procesmi, v dôsledku čoho paralelný program bude efektívnejší.
4. **Mapovanie** – priradenie zložených úloh z prechádzajúceho kroku na vykonávanie jednotlivým procesorom tak, aby boli všetky procesory maximálne využité a potreba komunikácie obmedzená na minimum.

Tento postup je znázornený na obrázku 18 a je známy ako **Fostrova metóda**.



Obr. 18: Dizajn paralelného programu podľa Fostrovej metódy

Rozdelenie

V tejto časti návrhu rozdelíme riešenie problému na veľké množstvo malých podúloh tak, aby mohli byť vykonávané v maximálnej možnej miere paralelne. Toto rozdelenie môžme prirovnať k jemnozrnnému paralelizmu. Neznamená to však, že tieto podproblémy budeme priamo riešiť, pretože by to vyžadovalo veľkú náročnosť na komunikáciu. Tiež musíme zobrať do úvahy aj špecifické vlastnosti architektúry paralelného výpočtového systému, na ktorom budú podúlohy riešené. Tieto skutočnosti neskôr zohľadníme v ďalších krokoch návrhu, napríklad pri aglomerácii znížime stupeň granularity, čo povedie k zväčšeniu jednotlivých podúloh a ich efektívnejšiemu vykonávaniu. Rozdelenie úlohy je možné urobiť najčastejšie na základe rozdelenia údajov, ktoré majú spracované alebo výpočtov, ktoré sa majú s nimi uskutočniť. Tento prístup odpovedá údajovej alebo funkcionálnej dekompozícii, ale nie je vylúčené použitie aj iného typu dekompozície [2].

Foster načrtol v práci [3] kontrolný zoznam, na základe ktorého je možné zhodnotiť správnosť realizácie krokov návrhu dizajnu paralelného programu. Krok rozdelenia pozostáva z nasledujúcich bodov:

Poznámka: V prípade použitia viacjadrových procesorov možno ako počet procesorov vziať počet všetkých dostupných jadier CPU.

- Počet vytvorených podúloh je aspoň o rád väčší ako počet procesorov, pre ktoré je program navrhovaný. V prípade nesplnenia tejto požiadavky hrozí riziko malej flexibility v ďalších krokoch návrhu paralelného programu.
- Rozdelenie by malo predchádzať redundantným výpočtom a ukladaniu údajov. Inak hrozí riziko, že program nebude škálovateľný do takej miery, aby bolo pomocou neho možné riešiť problémy s väčšou veľkosťou.
- Vytvorené podúlohy sú približne rovnako veľké. V prípade nesplnenia tejto požiadavky hrozí riziko zlého vyvažovania záťaže procesorov.
- Počet podúloh škáluje s veľkosťou problému. Inak program nebude schopný riešiť veľké inštancie problému s použitím väčšieho množstva procesorov.
- Sú vyskúšané rôzne alternatívy rozdelenia problému. Ich včasná identifikácia umožňuje väčšiu flexibilitu pri neskoršej voľbe vhodného návrhu.

Komunikácia

Podúlohy riešeného problému vytvorené podľa predchádzajúceho kroku sú navrhnuté tak, aby mohli byť vykonávané paralelne, čo však neznamená, že sú úplne nezávislé. Preto je potrebné zaoberať sa potrebou ich vzájomnej komunikácie. Jednotlivé podúlohy sú riešené v podobe procesov alebo vlákien a navzájom si vymieňajú čiastočné výsledky výpočtov alebo sa synchronizujú. V tomto kroku návrhu paralelného programu je potrebné určiť, aká komunikácia medzi procesmi je potrebná pri výmene údajov, a aké sú požiadavky na vhodný

komunikačný kanál medzi procesmi. Foster rozlišuje dva druhy komunikácie: lokálna a globálna. **Lokálna komunikácia** je realizovaná medzi malou podskupinou procesov pomocou posielania a prijímania správ medzi procesmi. **Globálna komunikácia** sa uskutočňuje medzi veľkým počtom procesov, ktoré spolu komunikujú. Príkladom takejto operácie môže byť zbieranie výsledkov jedným procesom od všetkých ostatných procesov. Pri použití globálnej komunikácie nedefinujeme vhodný komunikačný kanál, nakoľko v používaných jazykoch a knižniciach sú implementované vhodné nástroje pre takýto typ komunikácie. Pripomeňme si, že množstvo potrebnej komunikácie je potrebné čo najviac minimalizovať z dôvodu jej nákladov a zvýšeného času réžie paralelného programu. V tomto kroku kontrolný zoznam pozostáva z nasledujúcich bodov:

Poznámka: Pri implementácií globálnej komunikácie zväčša používame operácie pre kolektívnu komunikáciu.

- Každý z procesov komunikuje v približne rovnakej miere, v opačnom prípade nie je komunikácia rovnomerne rozdelená medzi procesormi.
- Každý z procesov komunikuje len s malou podskupinou susedných procesov.
- Operácie komunikácie sú vykonávané súbežne.
- Výpočty v jednotlivých podúlohách je možné vykonávať súbežne. V opačnom prípade je potrebné zvážiť preusporiadenie výpočtov a operácií komunikácie.

Aglomerácia

V predchádzajúcich krokoch sme sa snažili využiť maximálne dostupný výpočtový výkon rozdelením úlohy na čo možno najväčšie množstvo podúloh. Toto riešenie však v praxi väčšinou nevedie k efektívnemu paralelnému programu. Okrem toho vykonávanie každej z týchto podúloh ako samostatného procesu alebo vlákna by viedlo k neúmernému zvýšeniu réžie paralelizácie. Tiež väčší počet procesov ako procesorov, hoci už len o niekoľko rádov, vedie k ich vykonávaniu s pomocou multitaskingu, čo so sebou nesie ďalšie spomalenie.

Poznámka: Multitasking umožnuje zdanlivo súbežné vykonávanie viacerých úloh na princípe rýchleho striedania úloh. V jednom okamihu času je každé jadro procesora schopné spracovávať iba jedinú úlohu.

Pri aglomerácii sa zameriavame na to, či je možné niektoré podúlohy navzájom kombinovať, a tým znížiť počet potrebných podúloh na vyriešenie úlohy, pričom takto vytvorené podúlohy budú väčšej veľkosti. Tiež je potrebné preskúmať, či nie je užitočné niektoré údaje alebo výpočty replikovať. Zvyčajne je vhodné, aby počet takto vytvorených podúloh aglomeráciou mierne prevyšoval počet procesorov, čo poskytuje možnosť súbežnej komunikácie a efektívneho paralelného programu.

Poznámka: V prípade, že poznáme parametre cieľového paralelného systému, pre ktorý paralelný program píšeme, môže tomu prispôsobiť počet procesov, čo uľahčí ich následné mapovanie v ďalšom kroku.

Jedným z cieľov aglomerácie je zníženie komunikačných nákladov redukciou počtu podúloh. V prípade, že kombináciou niekoľkých podúloh do jednej väčšej sú údaje spracovávané v týchto podúlohách priamo dostupné v pamäti výslednej podúlohy a nie je potrebný ďalší prenos údajov medzi pôvodnými podúlohami. Ďalšiu úsporu je možné dosiahnuť agregáciou odosielaných správ do jednej väčšej správy, čím sa ušetrí čas potrebný z dôvodu latencie siete. Druhým z cieľov aglomerácie je zabezpečiť škálovateľnosť a prenositeľnosť navrhnutého

paralelného programu. Toho je možné dosiahnuť izolovaním dostatočného počtu podúloh. Ak by bol počet úloh vytvorených v aglomerácií príliš malý, viedlo by to k nízkej škálovateľnosti. To znamená, že paralelný program by nebolo možné vykonávať s použitím viacerých procesorov, ako pre koľko bol pôvodne navrhnutý. Odporúčanie je, aby bol počet podúloh vytvorených aglomeráciou o jeden rád vyšší ako počet použitých procesorov. Tretím cieľom aglomerácie je zníženie nákladov na vývoj paralelného programu. Pokiaľ pri návrhu paralelného programu vychádzame zo sériového programu je výhodné, pokiaľ z neho vieme čo najviac použiť a musíme vykonať čo najmenej zmien.

Kontrolný zozname pre krok aglomerácie pozostáva z týchto bodov:

- Aglomerácia znižuje náklady na komunikáciu tým, že vedie k vyššej lokálnosti údajov. Inak je potrebné preveriť, či tohto nie je možné dosiahnuť použitím iného postupu aglomerácie.
- Ak sme pri návrhu použili replikáciu určitých výpočtov vo viacerých procesoch, je potrebné preveriť, či je čas takto ušetrený väčší, ako čas potrebný na komunikáciu pri prenose týchto údajov.
- Ak sme pri návrhu použili replikáciu údajov, tak ich množstvo môže byť také veľké, aby neohrozilo škálovateľnosť a veľkosť problému, ktorý je možné paralelným programom riešiť.
- Podúlohy vytvorené aglomeráciou majú podobné výpočtové a komunikačné náklady.
- Počet vytvorených podúloh rastie úmerne s veľkosťou riešeného problému.
- Počet vytvorených podúloh aglomeráciou je čo najmenší, a pritom dostatočne veľký vzhľadom na počet procesorov cieľového paralelného systému. Stupeň súbežnosti paralelného programu je primeraný počtu použitých procesorov.
- Pomer medzi dekompozíciou a aglomeráciou je vyvážený a náklady na modifikáciu sériového programu neprevyšujú zisk z paralelizácie tohto programu.

Mapovanie

Posledným krokom potrebným k vyriešeniu úlohy pomocou paralelného programu je mapovanie alebo priradenie jednotlivých podúloh na vykonávanie jednotlivým procesorom. Pri použití multiprocesorov je toto mapovanie zabezpečené samotným operačným systémom. V systémoch s distribuovanou pamäťou je potrebné mapovanie zabezpečiť pri návrhu paralelného programu. Cieľom je priradiť procesy na vykonávanie tak, aby bol celkový čas výpočtu čo najkratší. Toho je možné docieliť priradením procesov na jednotlivé procesory, čo povedie k lepšiemu paralelizmu. Naopak podúlohy, ktoré navzájom často komunikujú by mali byť priradené tomu istému procesoru, čím sa zabezpečí lepšia lokálnosť údajov a zníži sa potreba komunikácie. Tieto dve požiadavky sú protichodné, a preto je potrebné nájsť medzi nimi akceptovateľný kompromis. Dosiahnutie čo najkratšieho času

vykonávania programu je možné dosiahnuť pomocou dobrého vyvažovania záťaže (viď. podkapitola 4) tak, že sa čo najviac minimalizuje čas, kedy procesory musia čakať a nevykonávajú žiadnu podúlohu.

Kontrolný zoznam pre tento krok pozostáva z nasledujúcich bodov:

- Za účelom minimalizácie komunikačných nákladov je potrebné zvážiť, či bude procesoru priradená iba jedná alebo viac podúloh.
- Analyzovať prínos statického a dynamického priraďovania procesov procesorom.
- V prípade použitia statického priraďovania sa uistiť, že počet procesov je o jeden rád väčší ako počet použitých procesorov.
- V prípade použitia dynamického priraďovania sa uistiť, že hlavný proces (master), ktorý zabezpečuje priraďovanie podúloh procesorom sa nestane úzkym miestom (bottleneck) paralelného programu.

Literatúra

[1] Timothy G Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Pearson Education, 2004 (cited on page 2).

[2] Zbigniew J. Czech. *Introduction to parallel computing*. UK: Cambridge University Press, 2016 (cited on pages 2, 9, 17, 21).

[3] Parallel Computing and I Foster. *Designing and building parallel programs*. http://www.mcs.anl.gov/~itf/dbpp. 1995 (cited on pages 2, 20, 21).

[4] Chuck L Lawson et al. "Basic linear algebra subprograms for Fortran usage". In: (1977) (cited on page 3).

[5] Jack J Dongarra et al. "An extended set of FORTRAN basic linear algebra subprograms". In: *ACM Transactions on Mathematical Software (TOMS)* 14.1 (1988), pp. 1–17 (cited on page 3).

[6] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010 (cited on pages 4, 14).

[7] Alessandro Fava, Emanuele Fava, and Massimo Bertozzi. "MPIPOV: a parallel implementation of POV-Ray based on MPI". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 1999, pp. 426–433 (cited on page 5).

[8] Bernd Freisleben, Dieter Hartmann, and Thilo Kielmann. "Parallel raytracing: a case study on partitioning and scheduling on workstation clusters". In: *Proceedings of the thirtieth hawaii international conference on system sciences*. Vol. 1. IEEE. 1997, pp. 596–605 (cited on page 5).

[9] Blaise Barney. *Introduction to Parallel Computing*. https://computing.llnl.gov/tutorials/parallel_comp/. 2019 (cited on page 8).

[10] Peter S. Pacheco. *An Introduction to Parallel Programming*. MA USA: Morgan Kaufmann, 2011 (cited on pages 12, 14).

[11] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485 (cited on page 12).

[12] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533 (cited on page 13).