Paralelní a distribuované algoritmy (PRL)

2020/2021

František Horázný – xhoraz02

Mesh multiplication

Úvod k algoritmu

Paralelní algoritmus mesh multiplication slouží k násobení dvou matic. Násobení matic není komutativní. Násobení matic A*B je definováno pouze pokud A je tvaru k*n a B je tvaru n*l. Výsledná matice má pak tvar k*l. Algoritmus využívá možnosti rozděli výpočet na jednotlivé buňky výsledné matice. Protože každá buňka je počítána jako násobek dvou vektorů (řádek z první matice a sloupec z druhé matice) a následná suma. Problém u tohoto algoritmu je pouze v distribuci dat. Nelze totiž všem procesorům zároveň dodat všechny potřebná data tak, aby se to časově vyplatilo. Je zřejmé, že pokud by to možné bylo, pak by doba výpočtu byla závislá pouze na n rozměru matic a složitost by tedy byla O(n).

Teoretická realizace algoritmu

Pro tento algoritmus je využito k*l procesorů, které jsou propojeny do mřížky k*l. Vstup celého systému jsou dvě matice k*n a n*l a výstupem je jedna matice k*l. Můžeme si zvolit jakýkoli procesor jako kořenový, protože zpomalení kořenového procesoru spočívá v distribuci dat krajním procesorům, ale počítadlo času spouštíme až po této akci. Já jsem i přes to volil poslední procesor, který je na počátku algoritmu nevyužíván a proto může rozesílat data ze souborů bez zpomalení.

Kořenový procesor načte soubory s maticemi. Zjistí rozměry výsledné matice a společný rozměr. Následně předá informace o rozměrech všem procesorům a rozešle data na levé krajní a horní krajní procesory. Dále pokračuje jako obecný x-tý procesor. Nakonec přijímá výsledné hodnoty od všech procesorů a složí výslednou matici.

Obecně x-tý procesor nejdříve přijme informace o rozměrech procesorové mřížky a zjistí svou polohu v ní. V cyklu, který je opakovaný n krát přijímá data od horního a levého souseda. Pokud tyto sousedy nemá, přijímá od kořenového procesoru. Hodnoty mezi sebou vynásobí a přičte do svého mezivýsledku. Pak hodnotu, která přišla od levého souseda pošle pravému sousedovi a hodnotu, která přišla od horního souseda pošle dolnímu sousedovi. Pokud některého z těchto sousedů nemá, dále data neposílá. Po dokončení cyklu odešle svůj mezivýsledek kořenovému procesoru.

Analýza složitosti algoritmu

Každý procesor počítá se složitostí *O(n)*, kde n je společný rozměr matic stejně jako výše. Neboli je to délka vstupních dvou vektorů do procesoru. Každý prvek postupně vezme, vynásobí a přičte do mezivýsledku. Celková složitost však musí také počítat se zpožděním způsobeným distribucí dat (kdyby každý procesor měl od začátku k dispozici rovnou všechny potřebná data, byla by složitost *O(n)*). Každý krok se data posunou od levého horního rohu nahoru nebo dolu. Je tedy zřejmé, že k poslednímu procesoru se data dostanou až za *k+l* „kroků“. Proto nemusíme počítat s oběma možnostmi, ale můžeme počítat s maticemi, kde *k=l* jako nejhorší možnost. Dále společný rozměr n lze zjednodušit na *n=k=l*, protože v ramci asymptomatické složitosti platí *O(n) = O(n+const)*. Tím se dostáváme na časovou složitost:

$$t(n) = O(k+l) + O(n) = O(2n) + O(n) = O(3n) = O(n).$$

Prostorová složitost je jednoduchá z definice algoritmu:

$$p(n) = n*n = O(n^2).$$

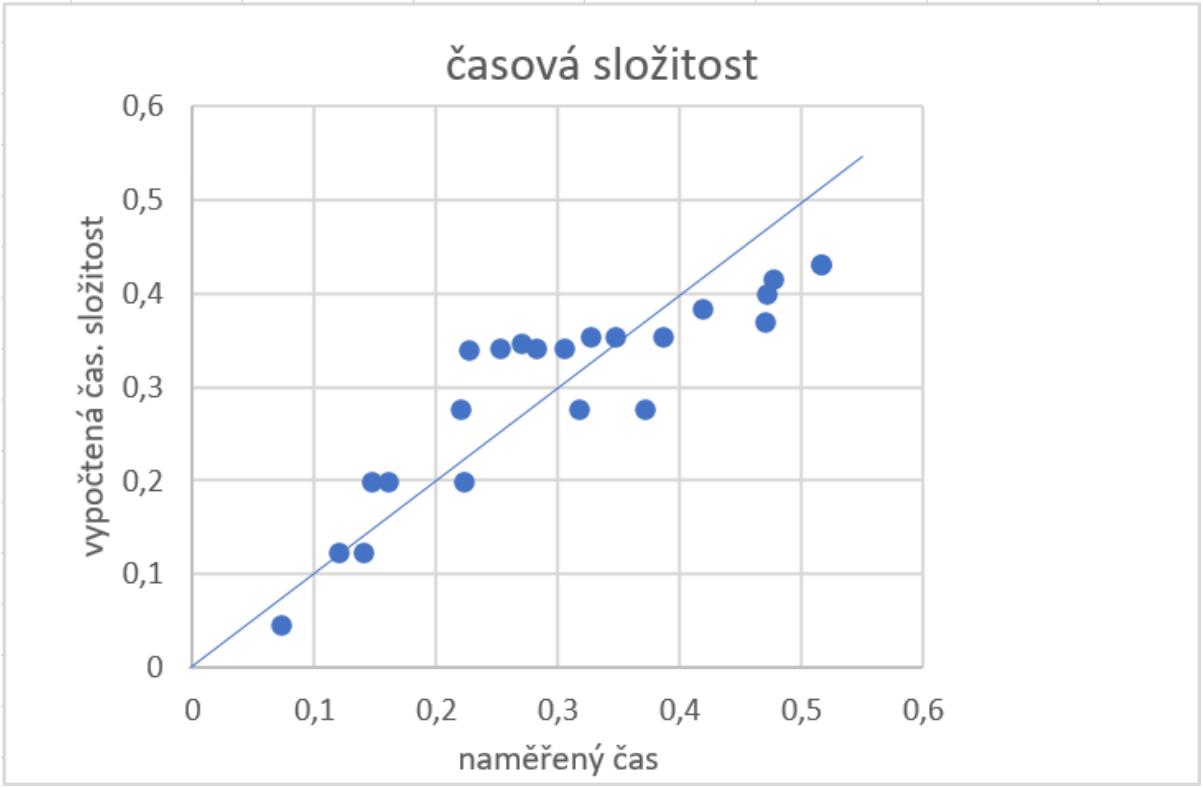
A z toho jednoduše vypočteme cenu:

$$C(n) = t(n) * p(n) = O(n) * O(n^2) = O(n^3)$$

Následující grafy by měli demonstrovat výše uvedenou složitost na reálných spuštěních programu.

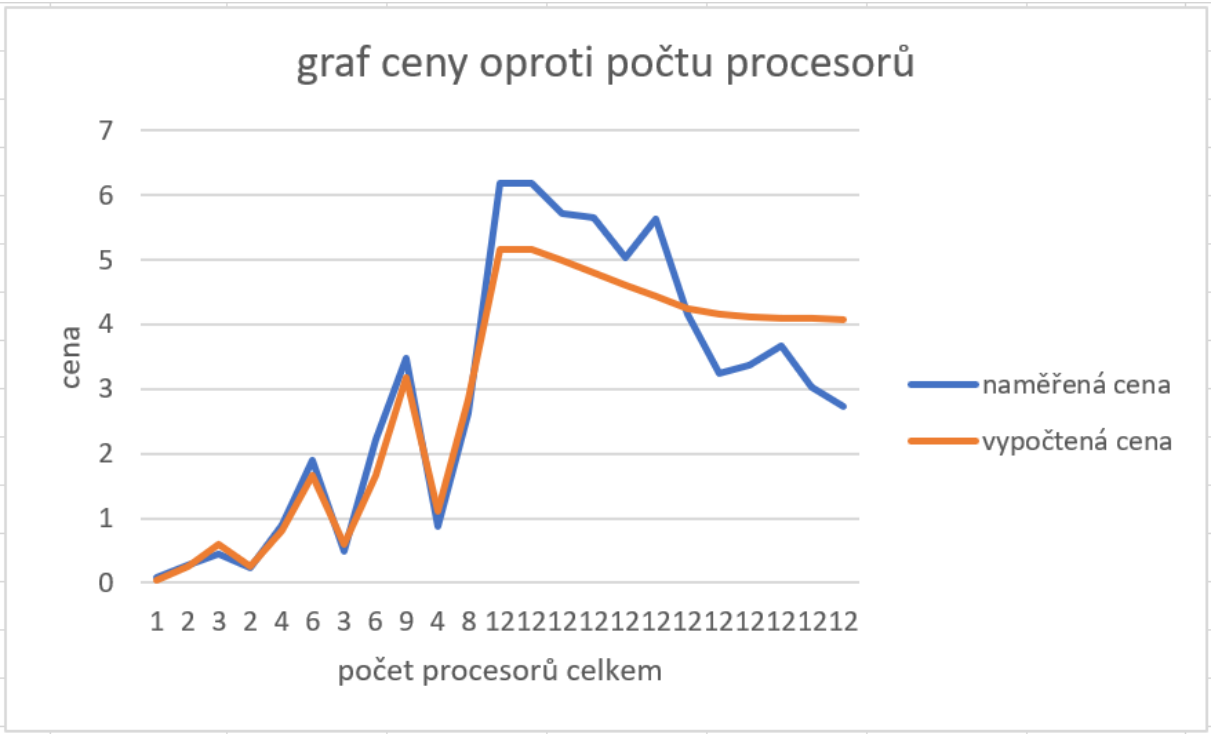
První graf ukazuje bodový graf mezi vypočtenou složitostí a naměřeným časem běhu algoritmu. Pokud jsou tyto hodnoty závislé, pak by se měli pohybovat blízko znázorněné přímce. Z teorie o asymptomatické složitosti víme, že složitost je možné násobit konstantou a přičítat konstantu. Měření probíhalo s různými hodnotami k l a n. Vypočtená složitost je tedy upravena následovně:

$$t(k,l,n) = (100k+100l+n)/1300 - 0,2$$



Další graf znázorňuje cenu vycházející z hodnot znázorněných v předchozím grafu. Tedy:

$$C(k,l,n) = t(k,l,n) * k*l$$



Tento graf má zvláštní tvar, protože vychází z diskrétních dat jednotlivých testovacích případů. Testovací případy byly dvou druhů:

Společný rozměr 120 a výsledná matice: 1x1, 1x2, 1x3, 2x1, 2x2, 2x3, 3x1, 3x2, 3x3, 4x1, 4x2, 4x3

Výsledná matice 4x3 a společný rozměr: 120, 100, 80, 60, 40, 20, 10, 5, 4, 3, 2

Implementace

Popis přesného provedení implementace by u tohoto algoritmu neměl smysl, protože kód je psán s komentáři a princip je relativně totožný s popsanou teoretickou realizací výše.

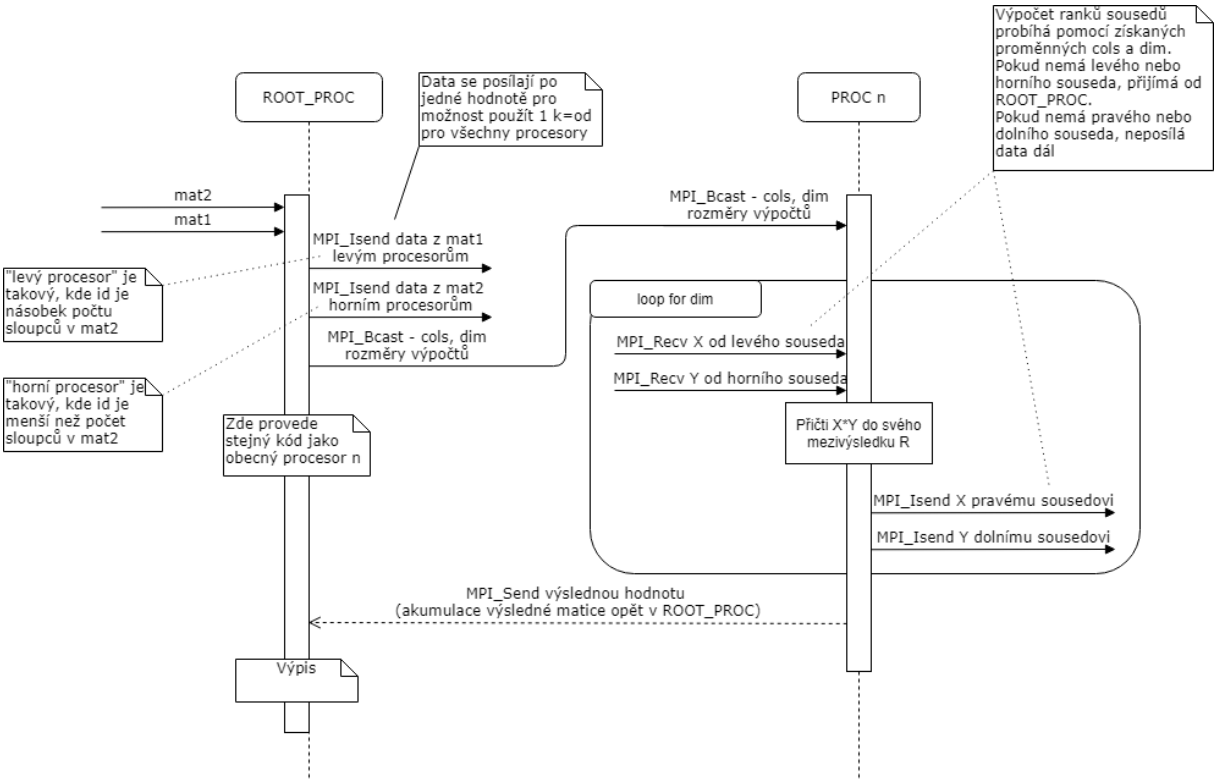
V implementaci jsem použil TAG, který však byl nutný pouze u rohového procesoru 0, který má levého i horního souseda totožného, ale musí vědět které dato má poslat kterým směrem.

Pokoušel jsem se naimplementovat testování rychlosti provedení přímo v kódu (vizte kód *main()*). Každopádně z mně nepochopitelného důvodu vždy první průběh běžel několikanásobně pomaleji než každý další. Proto tato feature nebyla využita pro výsledné měření rychlosti a kód byl vždy pouštěn s *#define TEST_COUNT 0*. Pro zajímavost jsem však tuto část kódu nemazal (protože mě stála kus práce).

Program nemá definované chování při nekorektním vstupu!!!

Komunikační protokol

Pro posílání dat využívám neblokující MPI_Isend. Hlavně z toho důvodu abych mohl data posílat po jednom čísle a nemusel psát kód pro různé lokace procesorů (pro krajní, rohové a procesory uprostřed). Zbytek komunikace je podrobně popsán v sekvenčním diagramu a poznámkách u něj.



Závěr

Myslím, že by bylo možné implementovat rychlejší a optimálnější verzi (například předpřipravit data na krajní procesory, rozdělit kód pro různé umístění procesorů nebo kontrolovat vstupní soubory a podobně). Mrzí mě, že jsem nepřišel na důvod, proč při běhu několikrát za sebou již v kódu *mm.cpp* je výpočet úplně jiné rychlosti. Každopádně jsem rád, že mi grafy složitosti vyšly velmi podobné složitosti teoretické.