1 Mesh multiplication

Zadáním projektu bylo implementovat pomocí knihovny Open MPI algoritmus Mesh multiplication. Celý program je spouštěn pomocí skriptu: `./test.sh` a soubory *mat1* a *mat2*, obsahující matice. Skript spouští `mm.cpp` spolu s $m\ x\ k$ procesory, kdy mat1 má rozměry $m\ x\ n$ a mat2 $n\ x\ k$.

2 Rozbor a analýza algoritmu

Algoritmus běží na $m\ x\ k$ procesorech, přičemž prvky matic A, B se přivádějí do procesorů pomocí 1. řádku a 1. sloupce procesorového pole. Každý procesor obsahuje proměnnou c, reprezentující prvek výsledné matice.
V algoritmu s jedním procesorem by se výsledná matice počítala po řádcích a sloupcích, ve dvou vnořených cyklech, zde jsou pak prováděny paralelně. Pro výpočet hodnoty c procesor čeká, až mu sousední procesory zašlou hodnoty *a, b*. Hodnotu a posílá procesor o sloupec před ním, hodnotu b procesor nad ním. Pokud není hraničním prvek, pošle *a, b* hodnoty svým následujícím sousedům (vedle a pod). To učiní tolikrát, kolikrát přijme hodnoty *a, b*, což je celkem *n-krát*.
Časová složitost $t(n) = m + k - 2$, což odpovídá lineární složitosti.
Počet procesorů $p(n) = O(n^2)$.
Cena $c(n) = p(n) * t(n) = O(n^3)$.
Algoritmus není optimální.

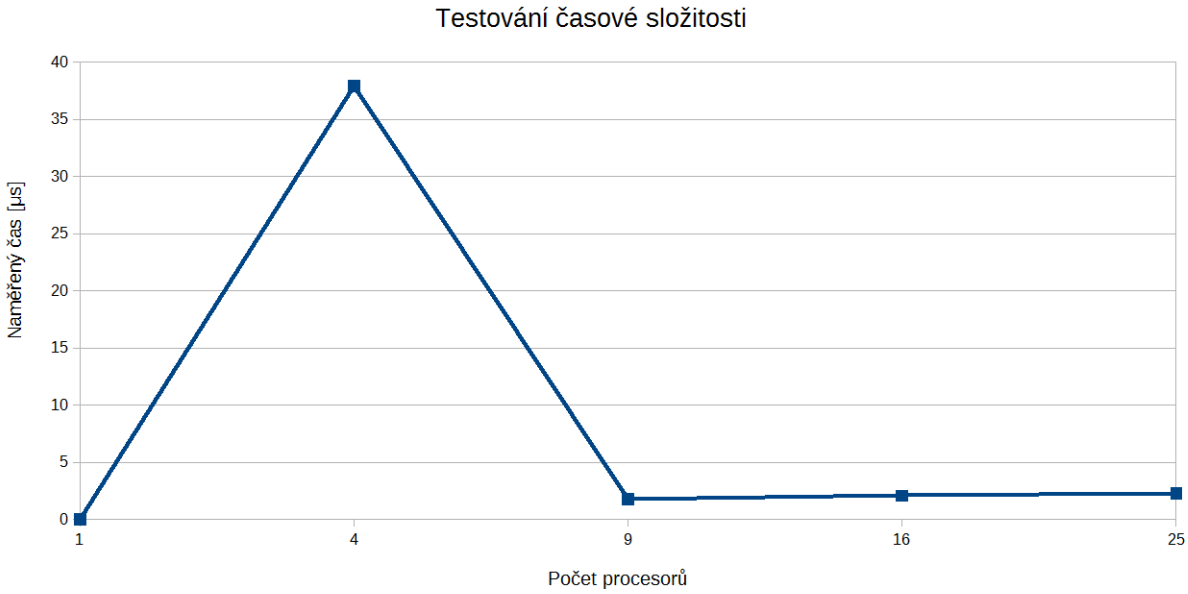
3 Implementace

V rámci volání skriptu `test.sh` probíhá předkontrola vstupu, během kterého jsou načteny řídící číselné hodnoty ze souborů – počet řádků u mat1 a počet sloupců u mat2.
Poté se zkontrolují i obracené hodnoty, abych se ujistila, zda je možné matice násobit. Tímto krokem implicitně ověřuji i existenci souborů a právo z nich číst.
Přestože jsem zde, na rozdíl od prvního projektu, neměla vyhrazený procesor pro řízení, procesor s id 0 se i zde stará o náležitosti k běhu programu navíc.
Nejdříve zpracovává vstup a načítá do poměti vstupní matice. Pokud v této části nastane chyba – například ve vstupní souboru není dost čísel, či se zde nachází nečíselné hodnoty, pošle procesor všem svým kolegům zprávu, že program nemůže být dokončen, načež se všichni ukončí.
V opačném případě odešle kopii matic prvnímu řádku a prvnímu sloupci procesorů výsledné matice.
Na konci také sesbírá výsledné hodnoty c, které dle požadavků vypíše na výstup.
Mezitím probíhá algoritmus samotný. Z podstaty problému víme, že počet dvojic hodnot *a, b*, se kterými procesor počítá je shodný s n. Proto jsem místo smyčky, čekající na příchozí hodnoty, implementovala for cyklus. V něm první řada a první sloupec procesorů postupně čtou hodnoty *a,b* ze svých pamětí a posílají se svým sousedům. Pokud procesor dostane tyto hodnoty, vypočte si aktuální c a pokud není hraniční v rámci pole, přepošle přijaté hodnoty zase dál. Trochu speciálně se řeší příjem u procesorů 1. řádku a 1. sloupce, kdy na některé hodnoty nečekají, protože je mají implicitně přiřazené.

4 Testování

Testování jsem spouštěla přes automatizované testy. Nejdříve jsem testovala správnost implementace algoritmu a porovnávala správnost výsledku.
Poté jsem testovala v rozsahu 1 až 25 procesorů (včetně), přičemž každý rozsah jsem testovala 10

krát. Testy jsem spustila celkově 3 krát. Časové hodnoty jsou zprůměrované – nejdřív průměr 10 testů a pak průměr 3 vzniklých hodnot na n_i počet procesorů.
Měření jsem prováděla pomocí *MPI_Wtime()*, kdy jsem měřila od začátku for cyklu, který mi počítá počet odeslaných dvojic hodnot *a,b*, až do ukončení tohoto cyklu, kdy pak následuje odeslání hodnot procesoru 0, který výsledek vypíše na výstup.
Výsledky odpovídají lineární závislosti, pouze při 4 procesorech jsem trvale získávala podivně vyšší čas, než u zbylých měřených hodnot.



Obrázek 1: Graf výsledků testování

5 Komunikační protokol

Obrázek 2: Sekvenční diagram pro n procesorů

6 Závěr

Znovu jsem pracovala s knihovnou Open MPI pro programování paralelních výpočtů a úspěšně jsem implementovala Mesh multiplication, včetně kontroly vstupu. Analýza i testování pak ukázaly, že algoritmus pracuje v lineárním čase.