УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

ИСПИТНИ РАД

Кандидат: **Марко Његомир**
Број индекса: sw-38-2018

Предмет: Паралелно програмирање
Тема рада: **Множење матрица помоћу серијских и паралелних алгоритама**

Ментор рада: **проф. др Мирослав Поповић**

Нови Сад, Јун, 2020.

SADRŽAJ

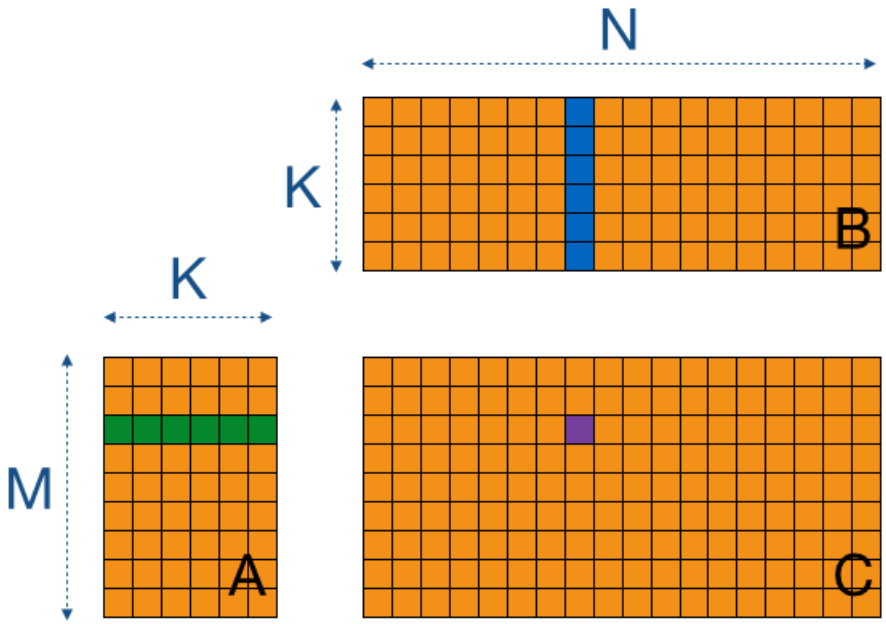
No table of figures entries found.

No table of figures entries found.

1. Увод

1.1 Проблем множења матрица

Множење матрица за циљ има да помножи две дате матрице, и да резултат смести у нову матрицу. Да би се матрице уопште могле помножити њихове димензије морају бити компатибилне, а то значи да број колона рве улазне матрице мора бити једнак броју редова друге улазне матрице. Димензије резултантне матрице зависе од броја редова прве улазне матрице, и броја колона друге улазне матрице.



Слика 1 Поступак множења матрица

1.1.1 Поступак решавања

Један једноставан приступ за множење матрица (Слика 1) је да се за сваки елемент матрице Ц који се налази у реду М и колони К пронађе исти ред у матрици А и иста колона у матрици Б, и затим се кореспондентни елементи пронађеног реда и колоне помноже, а затим се резултати множења саберу. Тај резултат се онда упише на одговарајуће место у матрици Ц.

Овакав начин решавања брзо постаје неефикасан са порастом димензија матрица па је због тога актуелно да се траже нови и побољшани начини да се овај поступак убрза.

Један од начина да се проблем матрица убрза је коришћењем паралелних алгоритама. То може бити паралелизован претходно описан поступак, или неки од напреднијих алгоритама као што су Страсенов алгоритам или Паралелни алгоритам за множење матрица по принципу подели-и-завладај.

У овом пројекту се анализира претходно описани паралелизовани поступак.

1.2 Задатак

Употребом TBB библиотеке реализовати алгоритам множења матрица целобројних вредности.

Потребно је реализовати:

- Серијску верзију програма за множење матрица
- Паралелну верзију алгоритма употребом TBB задатака и шаблона високог нивоа
- Скуп испитних случајева који потврђују исправност рада паралелне верзије

Матрице треба да се налазе у засебном директоријуму и да се учитају на почетку рада програма. Матрице могу али не морају бити квадратне – водити рачуна да димензије морају бити одговарајуће како би множење било дефинисано. Величину матрица треба варирати од малих до веома великих (у зависности од тога колико радна меморија дозвољава). За сваки тестни случај потребно је измерити време и упоредити серијску имплементацију, имплементацију уз помоћ parallel_for-а (користити све подразумеване вредности) и имплементацију уз помоћ TBB задатака.

Што се тиче самих TBB задатака, испитајте решење са различитом комплексношћу задатака. Један случај нека буде да сваки задатак рачуна само један елемент резултантне матрице. Други случај нека буде да сваки задатак рачуна целу једну врсту или колону у резултантној матрици. Трећи случај нека буде да сваки задатак рачуна 1/N елемената матрице где је N број језгара/хипернити доступних на вашем рачунару.

2. Анализа проблема

Серијско решење је веома једноставно за имплементацију, и може се урадити помоћу три угњеждене петље са по N итерација. Због тога се серијски алгоритам не скалира добро са порастом димензија матрица.

Паралелно решење имплементирано да одговара претходно поменутом серијском решењу има укупан рад $T_1(n) = \Theta(n^3)$, и распон $T_\infty(n) = \Theta(n)$. Када се рад подели са распоном добија се да је паралелизам $\Theta(n^2)$, што указује да ће паралелни алгоритам боље да се скалира са порастом димензија матрица од серијског.

Ако се резултати смештају у посебну матрицу, онда је задовољен Бернштајнов услов и тада нема трке до података па се због тога не мора водити рачуна о синхронизацији.

Чињеница да нема трке до података доста поједностављује имплементацију паралелног алгоритма помоћу **parallel_for** петље и помоћу **tbb** задатака, па се због смањених ограничења могу очекивати бољи резултати него када би се резултат множења матрица морао смештати у неку од полазних матрица.

3. Концепт решења

Да би се могле искористити неке од уграђених функција из стандардне библиотеке као што је функција **std::inner_product** која множи кореспондентне елементе 2 низа и затим их сабира, потребно је имплементирати матрице као једнодимензионе низове, јер се тада не морају правити помоћне структуре да би та функција исправно радила.

Због тога што се за рачунање једног елемента мора проћи кроз све колоне у датом реду прве матрице, и све редове дате колоне у другој матрици, кеширање података неће бити ефикасно због начина на који су подаци смештени у меморији.

Да би се кеширање побољшало потребно је транспоновати матрицу Б и изменити алгоритам множења тако да множи елементе једног реда матрице А са одговарајућим редом транспоноване матрице Б. Тада се сви подаци потребни за овакву операцију налазе на сукцесивним меморијским локацијама па је кеширање података ефикасније и има мање промашаја приликом читања из кеш меморије.

Због претходних напомена сви алгоритми за множење матрица за које се мери време извршавања користиће у програму транспоновану матрицу Б и функцију std::inner_product да би се добило најкраће време извршавања.

Решење је издељено на три програма:

1. Имплементација серијског алгоритма.
2. Имплементација паралелног алгоритма помоћу parallel_for петљи.
3. Имплементација паралелног алгоритма помоћу tbb задатака.

3.1 Концепт серијског алгоритма

Првобитно је имплементиран једноставан алгоритам за множење матрица, који множи матрицу А са матрицом Б и који се показао као доста спор.

Транспоновањем матрице Б алгоритам је постао три пута бржи, а додавањем inner_product функције време извршавања је затим преполовљено.

Додатне оптимизације у компајлеру су омогућиле пре свега бољу векторизацију функције inner_product па је време извршавања још једном преполовљено.

Овакав алгоритам је онда одабран да се пореди са паралелним алгоритмима.

3.2 Концепт паралелног алгоритма са parallel_for

Код паралелног алгоритма са parallel_for петљом је постојало више опција које могу утицати на време извршења неко код серијског алгоритма:

1. Транспоновање матрице Б
2. Коришћење функције inner_product
3. Одабир између три итерациона простора
    a. Blocked_range
    b. Blocked_range2d
    c. Blocked_range3d
4. Коришћење шаблонкласа за контролу поделе итерационог простора
    a. Auto_partitioner
    b. Simple_partitioner
    c. Affinity_partitioner
5. Подешавање вредности кванта поделе итерационог простора (grain size)
6. Угњеждавање паралелних петљи

Због исправности приликом поређења сви паралелни алгоритми за које се мери време транспонују матрицу Б и користе функцију inner_product.

Blocked_range3d се показао као бољи избор од ручног прављења угњеђдених parallel_for петљи, због тога што је tbb библиотека боље оптимизована за рад са тако дефинисаним итерационим простором.Али да би inner_product радио потребан је био blocked_range2d итерациони простор.

Комбинацијом транспоновања,blocked_range2d и inner_product функције добила се најбржа функција за множење матрица.

Она је још додатно убрзана експерименталним одређивањем кванта поделе итерационог простора. За blocked_range2d је потребно навести два кванта поделе, сваки за одговарајућу димензију. Показало се да је боље изабрати већи квант поделе за димензију која представља редове у матрици у односу на димензију која представља колоне у матрици. Однос кванта поделе за редове у односу на колоне се креће од 3/2 до 5/1 у завистности од димензија матрице и шаблонкласе за контролу поделе која се користи.

Слика 2 Приказ утицаја кванта поделе у односу на укупан рад који је представљен сивом бојом и исти је за обе фигуре

За матрице величине до 1000x1000 се користи affinity_partitioner јер се веће матрице теже могу сместити у бафер. Пошто се подаци за мање матрице могу сместити у кеш меморију, affinity_partitioner може да додели инсту итерацију петље оном језгру које је већ извршавало ту петљу, и тако се програм значајно убрзава, јер се искоришћавају "врући" подаци у кеш меморији.

Матрице веће од 1000x1000 користе ауто_партитионер и однос кванта поделе им је 5/1, и на тај начин се добије дупло краће време извршавања за велике матрице у односу на affinity_partitioner.

Обе наведене шаблонкласе за конторлу поделе итерационог простора су се показале ефикаснијим од simple_partitioner класе.

3.3 Концепт паралелног алгоритма са ТББ задацима

Код рада са ТББ задацима су одабрана три приступа:

1. Прављење задатака за рачунање свакок појединачног елемента излазне матрице.
2. Прављење задатака за рачунање целог реда излазне матрице.
3. Прављење онолико задатака колико има језгара у рачунару.

Пре почетка сваког задатка потребно је транспоновати матрицу Б и њу проследити коренском задатку. Време тог поступка се такође мери због доследности са другим алгоритмима, иако је само по себи незнатно у ондосу на укупно време извршавања.

3.3.1 Задатак за сваки појединачан елемент

Начин на који је ово имплементирано је да се направи један коренски задатак који затим створи по један задатак за сваки ред у матрици А. Сваки од тих задатака затим ствара по један нови задатак за сваку колону у матрици Б а који зна ком реду припада.

На тај начин се добије по један задатак за сваки елемент који је потребно израчунати.

Испоставило се да је овакав приступ прављења задатака у виду стабла бољи од директног прављења задатака за сваки елемент због тога што је време потребно за извршавање сваког задатка изузетно мало, па се код директног прављења много малих задатака често дешава покушај “крађе” задатака од других језгара. Показало се да се распоређивач задатака боље сналази при распоређивању када је овај проблем одрађен помоћу хијерархије задатака која овде личи на стабло са 3 нивоа.

3.3.2 Задатак за сваки појединачни ред

У овом приступу постоји један коренски sадатак, који направи нови задатак за сваки ред. Затим се рачунају сви елементи тога реда помоћу серијског алгоритма који користи транспоновану матрицу и inner_product функцију.

Оваково распоређивање се показало као јако ефикасно због тога што сада сваки задатак има више посла који мора да обави. Тако да је цена прављења задатка релативно мала у односу на количину посла који се обавља, што није био случај код алгортима који ствара задатак за сваки појединачни елемент.

Чињеница да се један исти ред у задатку множи са одговарајућим колонама омогућава ефикасније кеширање података за језгро које преузме овај задатак.

3.3.3 Прављење задатака сразмерно броју језгара

Овај приступ је веома сличан претходном по томе што се такође прави један коренски задатак, који онда прави задатке за редове. Разлика је у томе што се овде не прави задатак за сваки појединачни ред, већ се прави задатак за онолико редова колико одреди формула **број_елемената_реда/број_језгара**.

Идеја је да свако језгро добије значајан обим посла, и да се подаци над којима ради једно језгро нађу у кеш меморији дуже времена, као и да се смањи преузимање задатака од других језгара.

Овај приступ се у тестирањима показао као доста ефикасан, и у рангу је прављења задатака за сваки ред.

3.4 Одабир компајлера и опције компајлера

За овај пројекат сам одабрао да користим Intel C++ compiler v18. Опције које он нуди су скратиле време извршења неких алгоритама и до два пута.

- Функција **std::inner_product** може јако добро да се векторизује. Због тога се као опција у компајлеру може одабрати сет инструкција **AVX2** који омогућава да се за векторизацију користе **256bit** регистри. Компајлер који долази уз Visual Studio 15 има подршку само до SSE 4 скупа инструкција, и користи максимално 128bit регистре, па је извршавање на Интеловом Це++ компајлеру са овом опцијом скоро дупло брже. То је и због тога што је већина посла коју алгоритми обављају заправо рад inner_product функције, па се она дупло боље векторизује за довољно велике вредности матрица.
- Интелов компајлер нуди и O3 ниво оптимизације који је овде укључен.
- Постоји и опција за loop unroling која се може ручно подесити. На тестном рачунару је ова функција понекад правила проблеме па није додатно подешавана.
- Интел нуди и опције да оптимизује код за циљану архитектуру.
- Постоји и подешавање да се код оптимизује за Це++ 11
- Постоји још много других опција које имају мањи ефекат од наведених и неће бити наведене овде.

4. Опис решења

Решење садржи три пројекта. Сваки пројекат представља по један приступ решавања проблема множења матрица. Сви пројекти деле одређене сличности у структури и току извршавања. Овде ће бити наведене основне информације о структури и току извршавања а детаљнији описи функција са свим параметрима,повратним вредностима и грешкама су приложени у HTML формату уз пројектно решење.

4.1 Серијски програм - модули и основне методе

4.1.1 Модул главног програма (MainProgram)

```
int mullSerial(int argc, char* argv[]);
```

Почетна функција програм која у зависности од броја прослеђених аргумената позива различите функције. Валидност сваког множења се проверава аутоматски, тако што се пореди са тачним вредностима из сачуване датотеке.

- Ако аргументи нису прослеђени, ова функција учитава предефинисане матрице и мери време за сваку од њих тако што узима просечно време од 5 итерација. Резултати се исписују на конзолу у виду табеле резултата и резултат сваког множења се чува у одговарајућем фолдеру.
- Ако су прослеђени путање прве матрице, друге матрице и путања где се чува резултат онда се позива метода која рачуна само резултат множења те две прослеђене матрице уз исти наставак тока као кад нема аргумената.

4.1.2 Модул за рад са матрицама (MyMatrix)

Овај модул садржи све функције везане за множење матрица.

Ту спадају функције за:

- Серијско множење матрица
- Транспоновање матрица
- Учитавање матрица
- Прављење тестних података
- Чување резултата множења
- Исписивање матрица

4.2 Паралелни програм са parallel_for петљом

4.2.1 Модул главног програма (MyParallelProgram)

```
int mullParallel (int argc, char* argv[]);
```

Почетна функција програм која у зависности од броја прослеђених аргумената позива различите паралелне функције. Валидност сваког множења се проверава аутоматски, тако што се пореди са тачним вредностима из сачуване датотеке.

- Ако аргументи нису прослеђени, ова функција има исти ток и позива паралелне верзије метода које и серијски алгоритам позива.
- Ако су прослеђени путање прве матрице, друге матрице и путања где се чува резултат онда се позива метода која рачуна само резултат множења те две прослеђене матрице уз исти наставак тока као кад нема аргумената.

4.2.2 Модул за рад са матрицама (MyParallelMatrix)

Овај модул садржи све функције везане за паралелно множење матрица.

Ту спадају функције за:

- Паралелно множење матрица
- Транспоновање матрица
- Учитавање матрица
- Чување резултата множења
- Исписивање матрица

За паралелно множење матрица се користе функтори који обављају задати посао.

Постоје верзије функтора које раде са обичним итерационим простором, дводимензионим итерационим простором, тродимензионим итерационим простором, регуларним или транспонованим матрицама, матрицама које серијски рачунају коначну вредност елемента или је рачунају помоћу inner_product функције.

Време се мери за функцију која користи дводимензиони итерациони простор, транспоновану матрицу и inner_product функцију.

4.3 Паралелни програм за ТББ задацима

4.3.1 Модул главног програма (MainTaskProgram)

```
int mullTask (int argc, char* argv[]);
```

Почетна функција програм која у зависности од броја прослеђених аргумената позива различите паралелне функције. Валидност сваког множења се проверава аутоматски, тако што се пореди са тачним вредностима из сачуване датотеке.

- Ако аргументи нису прослеђени, ова функција учитава матрице предефинисаних димензија, и затим за сваки пар матрица мери време за три посебна алгоритма
    - Алгоритам који прави задатке за сваки елемент
    - Алгоритам који прави задатке за сваки ред
    - Алгоритам који додељује број редова сваком задатку сразмеран броју језгара у рачунару

  Сви резултати мерења се исписују кориснику на конзолу, а резултати сваког множења се чувају у посебним фолдерима одређеним за сваки тип алгоритма.
- Ако су прослеђени путање прве матрице, друге матрице и путања где се чува резултат онда се позива метода која рачуна само резултат множења те две прослеђене матрице уз исти наставак тока као кад нема аргумената.

4.3.2 Модул за рад са матрицама (MyTaskMatrix)

Овај модул садржи све функције везане за паралелно множење матрица уз помоћ тбб задатака.

Ту спадају функције за:

- Паралелно множење матрица уз помоћ тбб задатака
- Транспоновање матрица
- Учитавање матрица
- Чување резултата множења
- Исписивање матрица

Суштински је то исти модул као и код пројекта са parallel_for петљом.

4.3.3 Модул за рад са задацима(MullTask)

У овом модулу се налазе класе које представљају тбб задатке.

За сваки тип алгоритма постоји коренски задатак који кад се покрене даље ствара друге задатке у зависности од типа алгоритма:

- MullSingleElementTask - Користи се као коренски задатак који ствара задатке за сваки ред у матрици а они даље стварају задатке за сваку колону у том реду и на тај начин се добије задатак за сваки елемент матрице.
- MullSingleRowTask - Ако се позове као коренски задатак онда ствара задатке за сваки ред, који онда даље серијски рачунају резултате.
- MullDistributedTask - Овај задатак ствара онолико задатака колико има језгара у процесору, и сваком задатку је додељем број редова по формули број_елемената_у_реду/број_језгара. Разлог зашто није изабрано да се додељује подматрица димензија 1/H је зато што је структура матрице таква да ће се боље векторизовати петље ако се деле по редовима.

5. Тестирање

Након сваког рачунања множења матрица, валидност решења се аутоматски проверава поређењем са претходно сачуваним серијским решењем.

Постоји одређен скуп предефинисаних матрица које се аутоматски покрећу када није прослеђен ниједан аргумент, и за њих се рачунају времена извршавања, а након сваког израчунавања се проверава и валидност решења.

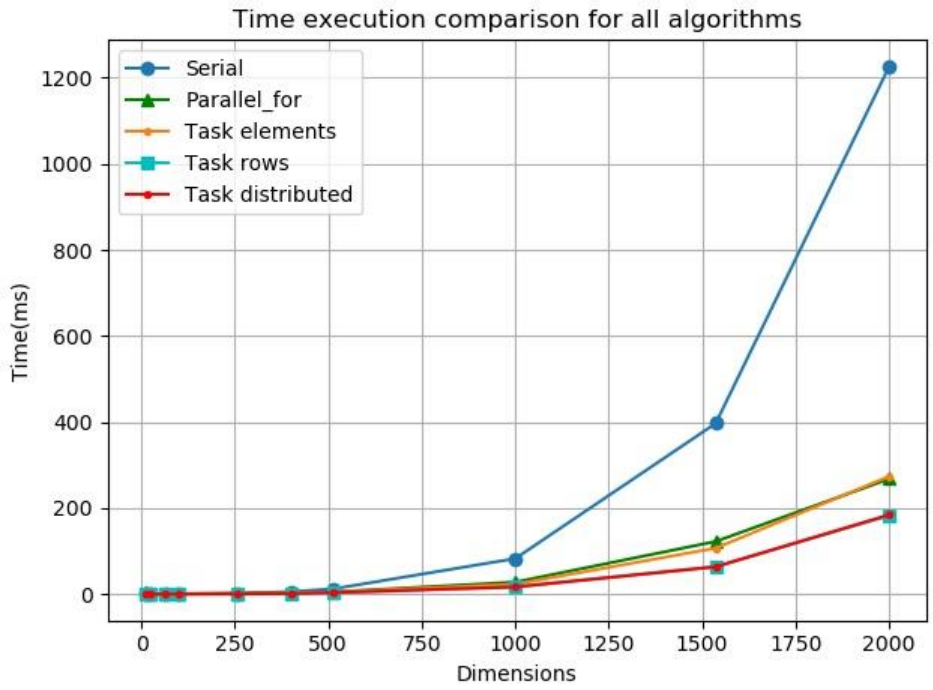
5.1 Спецификације рачунара

- Windows10
- Intel i7 9750h процесор (6 cores, 12 logical) @2.6GHz (4.5GHz boost)
- Nvidia GTX1660TI графичка картица
- 16GB DDR4 рам меморије (dual channel)
- 512 gb SSD

5.2 Резултати и анализа времена извршења

Algorithm\Size	10x10	25x25	64x64	100x100	256x256	400x400	512x512	1000x1000	1536x1536	2000x2000
Serial	0.00146	0.0056	0.04082	0.17838	1.34506	5.55838	11.8823	82.5212	396.866	1226.48
Parallel For	5.563	0.02446	0.24104	0.54884	1.86144	3.81368	4.99116	27.8248	122.319	267.762
Tasks Element	3.628	0.04552	0.1805	0.26678	1.13956	2.38504	4.5916	23.0061	106.817	273.628
Tasks Rows	0.007	0.01904	0.06882	0.16626	0.56024	1.52524	2.96602	16.7629	63.5601	182.583
Tasks Distributed	0.01168	0.02534	0.05072	0.13556	0.4876	1.28468	3.44048	16.5633	63.4098	184.405

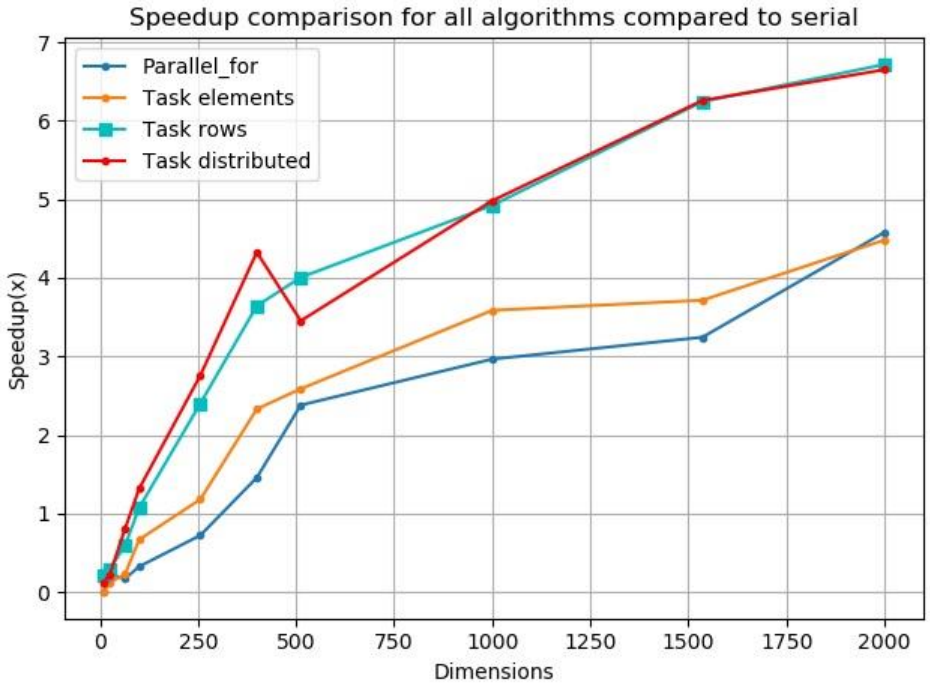
Error! Reference source not found. Прикупљени подаци о брзини извршења алгоритама.



Слика 3 Брзина извршења свих алгоритама

На графику се могу уочити три групе по брзини извршавања:

1. Серијски алгоритам који је сам и најспорији осим матрице веома малих димензија. То је због тога што је цена паралелизације и прављења задатака релативно велика у односу на количину посла који треба да се одради.
2. Parallel_for и тбб задатак за сваки елемент који су незнатно спорији од осталих тбб задатака
3. Тбб задаци са редовима и са прављењем задатака у односу на број језгара који су најбржи и између њих скоро па да и нема разлике



Слика 4 Убрзање алгоритама у односу на серијски алгоритам

На овом приказу се јасније види да убрзање расте у односу на серијски алгоритам са порастом димензија матрице.

Због великог могућег паралзима програма које сам раније поменуо овај проблем се добро скалира на више језгара. Има довољно посла да се сва језгра упосле. На рачунару на ком је тестирано са 6 физичких језгара, и 12 логичких није очекивано да се добије 12 пута обрзање због тога што та језгра деле ресурсе, али логичка језгра помажу да се пребаци та горња граница убрзања коју би имао процесор са само 6 језгара.

Занимљив је пад убрзања алгоритма који прави задатке зависно од броја језгара за димензије матрице 512x512. Могуће је да је то последица векторизације, јер је број 512 упола мањи од 256bit регистара које користе AVX2 инструкције, па су други алгоритми боље искористили векторизацију за те димензије, јер је приметно убрзање других алгоритама у тој тачки.

6. Закључак

Резултати добијени тестирањем потрврђују да је концепт решења добар и да је теоријска основа на којој се заснива концепт решења исправан. Добијене су очекиване вредности за убрзање уз веома мале аномалије.

Даља побољшања би се могла добити коришћењем Страсеновог алгортима или алгоритма који множи матрице по принципу подели-и-завладај, или једноставно повећањем броја расположивих језгара јер је паралелизам за овај проблем доста велики.

Као најбитнији фактори који утичу на брзину су се истакли:

- Коришћење AVX2 инструкција
- Транспоновање матрица
- Коришћење интеловог компајлера
- Коришћење inner_product функције која ради слично као map_reduce.
- Стварање тбб задатака уз присуство хијерархије задатака(налик на стабло)

Све наведене технике се јако добро комбинују и имају добру синергију, па су и допринеле томе да добијем што краће време извршавања код свих алгоритама, што ми је и био циљ.

Али постоји још доста начина да се овакви алгоритми убрзају а који овде нису тестирани, што показује и број радова на ову актуелну тему, и то оставља простора за даља тестирања.