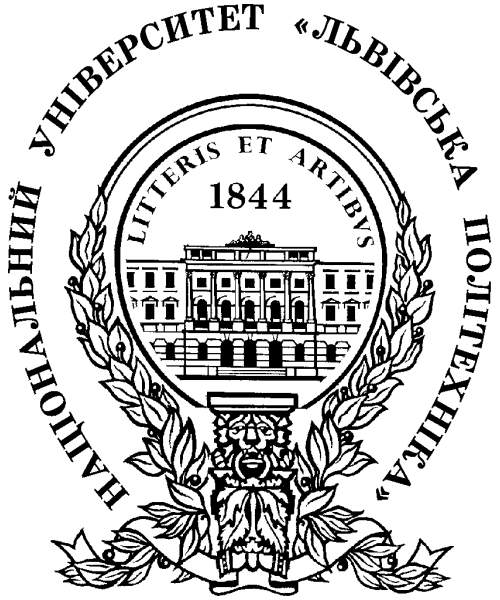
Міністерство освіти і науки України
Національний університет "Львівська Політехніка"
Інститут комп'ютерної техніки, автоматики та метрології

Кафедра ЕОМ



Реферат
на тему: «Програмні засоби паралельних обчислень в області лінійної алгебри»

Виконав:
студента групи СПРм-22
Лисенко С. В.
Прийняв:
Бочкарьов О.Ю.

Львів
2015

Зміст

1 Вступ

У даній роботі досліджено існуючі методи паралельного розв'язання задач лінійної алгебри та зроблено аналіз нових, перспективних напрямків дослідження у даній сфері. Також автором запропонований новаторський підхід до розв'язання вказаних задач.

Лінійна алгебра є надзвичайно цінним математичним інструментом, що знаходить застосування у таких розділах як фізика, криптографія, обробка сигналів, економіка, біологія, фармацевтика, дослідження операцій та багатьох інших. Лінійні моделі всеможливих систем мають простий опис. Володіння цим апаратом у сучасному світі є критичним для інженерів, а прискорення обчислень може мати прямий економічний ефект, оскільки перед сучасною індустрією ставлять все більші задачі, розв'язок яких може займати десятки і сотні днів на комп'ютерах, вартість яких обчислюється мільйонами, то дослідження програмних засобів розпаралелення задач лінійної алгебри є актуальним напрямком.

2 Аналіз методів розпаралелювання обчислень стосовно до лінійної алгебри

Сучасні застосування лінійної алгебри вимагають виконання операцій над багатьма змінними. Зростання розміру задач вимагає розробки швидших та дешевших шляхів їх розв'язання, що зазвичай робиться через розпаралелювання обчислень. Спеціальні програмні бібліотеки Basic Linear Algebra Subprograms (BLAS) реалізують поширені операції над щільними векторами та матрицями. Швидкі обчислення у лінійній алгебрі грають вирішальну роль у багатьох наукових застосуваннях. Часто виробники апаратних засобів також надають оптимізовані реалізації BLAS для своїх клієнтів.

2.1 Апаратний паралелізм

Ранні спроби використання паралельного виконання були через Single Instruction Multiple Data (SIMD) Паралелізм. Цей підхід використовує вектори даних у якості операндів та одночасно застосовує одну й ту саму операцію до кожного елементу вектора. Головним чинником, що обмежує SIMD паралелізм є пропускна здатність пам'яті.

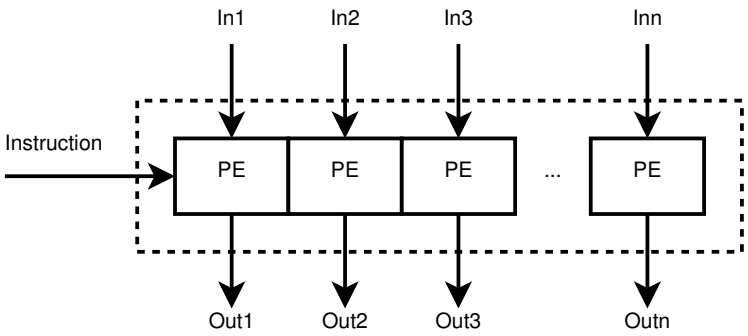


Рис. 1: Приклад SIMD архітектури

Сучасні суперскалярні архітектури реалізовують багато рівнів паралелізму, у тому числі паралеьне киконання окремих інструкції за допомогою конвеєра та векторизацію арифметичних операцій. Використання

більш ніж одного арифметико-логічного пристрою та невпорядкованого виконання інструкцій (Out-of-order execution) збільшує кількість команд, що одночасно можуть бути виконані. Часто розпаралелювання досягається через оптимізації компілятора та/або спеціального блоку процесора. У таких випадках програміст може не бути знайомим з деталями мікроархітектури процесора та не розпаралелювати код явно. Окремий тип архітектур процесора, що називається Very Long Instruction Word (VLIW), вимагає явного вказування інструкцій, що мають бути виконані паралельно.

Multiple Instructions Multiple Data (MIMD) системи мають набір незалежних процесорів, що можуть мати спільний адресний простір (системи зі спільною пам'яттю) або кожен свій окремий (системи з розподіленою пам'яттю). Кожен процесор розв'язує частину великої задачі, виконуючі свій власний набір інструкцій. инхронізація між процесорами реалізовується або через м'ютекси у системах зі спільною пам'яттю, або через обмін повідомленнями через комп'ютерну мережу для систем з розподіленою пам'яттю. Для систем з розподіленою пам'яттю ціна обміну повідомленнями зазвичай набагато більша за ціну обчислень. Це призводить до того, що розроблюються спеціальні алгоритми, що уникають комунікацій (communication avoiding алгоритми).

Системи зі спільною пам'яттю важко піддаються масштабуванню. Швидкість спільної пам'яті обмежує кількість процесорів, що можуть її ефективно використовувати. Пам'ять може бути реалізована таким чином, що доступ до окремих її блоків буде незалежним та одночасним. Така пам'ять називається багатоканальною. Наприклад сопроцесори Intel Xeon Phi мають 16-канальну пам'ять та 60 процесорів.

Здатність до масштабування систем з розподіленою пам'яттю залежить від типу задач, що розв'язуються. Для задач, що не вимагають синхронізації у процесі обчислень (shared nothing задачі) масштабування практично не обмежене. Для деяких задач може бути необхідна передача даних від кожного процесора до кожного на кожній ітерації алгоритму. У цьому випадку латентність та пропускна здатність мережі грає вирі-

шальну роль. У будь якому випадку одна ітерація алгоритму не може бути виконана швидше, ніж буде передане щонайменше 1 повідомлення через мережу.

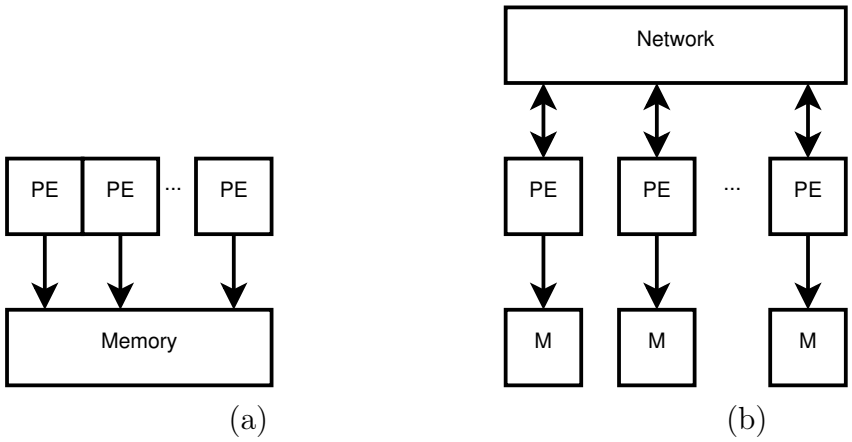


Рис. 2: Моделі комп'ютерів зі спільною(a) та розподіленою(b) пам'яттю

Non-Uniform Memory Access(NUMA) системи використовують неявну передачу повідомлень для доступу до пам'яті іншого процесора. Цей тип архітектур має більшу здатність до масштабування ніж прості системи зі спільною пам'яттю, оскільки для комунікації можуть бути використані спеціальні шивдкі шини з малою латентністю. Проте фізичні обмеження не дозволяють з'єднувати так багато процесорів, як у системах з розподіленою пам'яттю. Продуктивність NUMA систем сильно залежить від шаблонів доступу до пам'ятті, оскільки доступ до пам'ятті іншого процесора має значно більшу латентність.

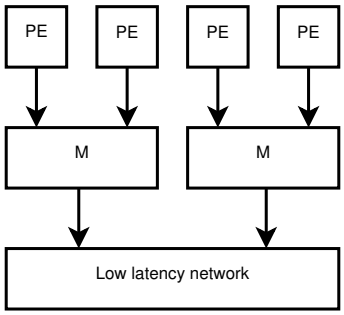


Рис. 3: Модель NUMA комп'ютера

2.2 Програмний паралелізм

У протилежність до паралелізму на апаратному рівні, програмний паралелізм досягається через реалізацію конкретних паралельних алгоритмів. Було розроблено декілька шаблонів паралельного програмування. Зазвичай при розробці паралельних алгоритмів має бути взята до уваги апаратна архітектура комп'ютера.

2.2.1 Fork-join паралелізм

Інший рівень паралелізму, Single Program Multiple Data (SPMD), став популярним для багатопроцесорних систем. Кожен процесор має свій власний простір виконання, проте виконує одну й ту саму програму. Кожен процесор приймає власну частину вхідних даних та обчислює частину результату. Цей підхід до розпаралелювання також відомий як fork-join паралелізм.

Fork-join шаблон проектування розбиває задачу на множину менших та незалежних задач (fork-стадія). Кожна мала задача далі розв'язується паралельно окремим процесором, після чого розв'язки маленьких зачад збираються назад у одне ціле (join-стадія).

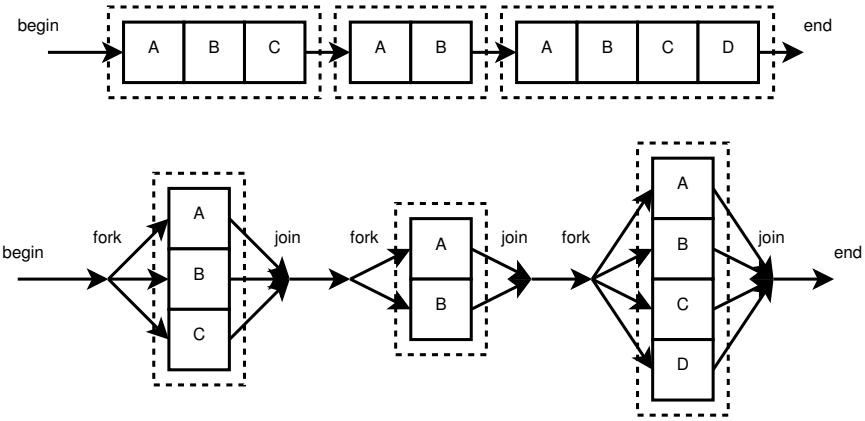


Рис. 4: Приклад Fork-join потоку виконання

Fork та join стадії є синхроними, тобто вони починають виконання одночасно та жоден процесор не може перейти до виконання інших задач

поки кожен з процесорів не виконає join стадію. Код, що розпаралелюється та послідовний код не може бути виконаний незалежно. Це призводить до значного погіршення продуктивності у випадках, коли послідовні та паралельні частини чередуються багато разів та кожна паралельна стадія є не великою для ефективного виконання на великій кількості процесорів. Fork та join стадії можуть мати векику вартість комунікацій у системах з розподіленою пам'яттю, оскільки данні мусять бути кожен раз передані по мережі.

Fork-join паралелізм широко використоується у таких бібліотеках як ScaLAPACK, паралельні реалізації BLAS та LAPACK. Алгоритм SUMMA для загального множення матриць на комп'ютерах з розподіленою пам'яттю, що використовується у ScaLAPACK для одного й того самого розміру матриць показує зничення продуктивності коли кількість процесорів, а відповідно і ціна комунікацій зростає.

Були розроблені кращі алгоритми, такі як CAPS[1], проте вони можуть мати суттєві обмеження на кількість процесорів або розмір матриць(Для CAPS розмір матриць має бути $2^N \times 2^N$ блоків та кількість процесорів має бути 7^i). Такі алгоритми не можна розглядати у якості загальних алгоритмів для множення будь-яких матриць.

Ця модель розпаралелювання вимагає синхнонізації перед комною fork та join стадією. Над даними у рамках паралельного обчисленя можна виконувати лише один і той самий набір інструкцій. Також для багатьох алгоритмів існують обмеження на кількість частин, на які можна розбити задачу. ScaLAPACK використовує блочно-циклічне розподілення матриць. Це вимагає, щоб кількість процесорів було добутком 2 чисел $P_M \times P_N$. Вкрай бажано, щоб ці числа були якомога ближчі одне до одного, оскільки це значно впливає на продуктивність.

2.2.2 Паралелізм рівня задач

Такі бібліотеки як PLASMA[3] та FLAME[2] використовують паралелізм рівня задач. Метод невпорядкованого виконання інструкцій (Out-of-order execution) викоритовується для паралельного виконання незалежних ко-

манд. Цей метод добре відомий та використовується у суперскалярних процесорах для реалізації паралелізму рівня інструкцій.

Матриці розділяються на блоки та операції над матрицями представляються як набір операцій над блоками. Кодна операція розглядається як атомарна задача(Task). Кожна задача має свої вхідні та вихідні дані. Обчислення складаються з двох стадій:

1. Послідовна побудова списку задач.

2. Невпорядковане паралельне виконання задач зі списку.

Деякі задачі можуть мати залежності по даних між собою(так звані Data Hazards). Ці залежності обмежують кількість способів виконання задач. Розглядаються 3 типи залежностей по даних:

1. Read after write залежність – Задача B використовує данні у якості вхідних, коли задача A використовує їх у якості вихідних.

2. Write after read dependency – Задача B використовує данні у якості вихідних, коли задача A використовує їх у якості вхідних.

3. Write after write dependency – Задачі A та B використовую одні й ті самі данні у якості вихідних.

Задача B була додана до списку пізніше ніж задача A. У кожному з випадків виконання задачі B має початись лише після фів виконання задачі A.

Задачі представляються у вигляді арієнтовного ациклічного графа(directed acyclic graph, DAG). Кожна вершина представляє собою задачу. Ребрами представлені залежності по даних. Незадовільненими називаються залежності, що виражають залежність однієї задачі від ішної, ще не виконаної задачі. У кожен момент часу може бути багато задач без незадовільнених залежностей. Такі задачі можуть бути виконані одночасно.

Перевагою такого медоту є те, що окремі незалежні задачі пожуть містити відмінні операції над даними. Наприклад загальне блочне множення матриць вимагає 2 типи операцій: множення та додавання блоків

матриць. Ці операції можуть не мати явних залежностей між собою, проте при використанні fork-join паралелізму виконуються послідовно.

Також ефективність DAG паралелізму менше залежить від кількості процесорів. Цей підхід реалізований у проектах FLAME та PLASMA. Мета цих проектів – сховати паралелізм рівня задач за інтерфейсом, що схожий з LAPACK. PLASMA ніколи не використовує DAG явно. Задачі компактно зберігаються у орієнтованому списку. Нові задачі додаються у кінець списку. DAG планувальник проходить по списку з початку та обирає незалежні задачі для виконання.

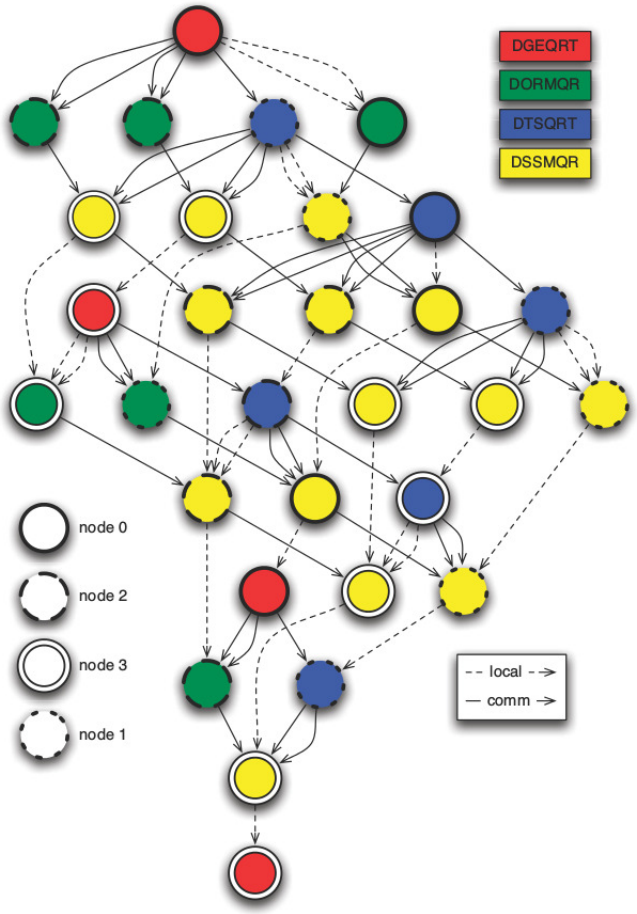


Рис. 5: DAG для QR факторизації 4×4 блочних матриць[5]

Останні дослідження доводять, що DAG паралелізм може мати набагато більшу продуктивність ніж традиційний fork-join підхід[2, 3, 4, 5].

QUARK Проект – це спроба полегшити паралельне програмування

з використанням паралелізму рівня задач, надаючі простий інтерфейс для побудови задач та залежностей по даних між ними та середовище паралельного виконання задач.

Реалізація блочних алгоритмів з лінійної алгебри з використанням QUARK паказує приблизно ту саму продуктивність, що має PLASMA. Оскільки QUARK використовує явний паралелізм, він може у деяких випадках показувати більшу продуктивність, ніж це робить PLASMA. У PLASMA кожен виклик підпрограми є синхроним, тобто окремі підпрограми виконуються послідовно. у QUARK можливе їх паралельне виконання. Часто розв'язання системи лінійних рівнянь $Ax = b$ досягається через LU факторизацію та наступним розв'язанням методом Гауса.

$$LU = A,$$
$$y = L^{-1}b,$$
$$x = U^{-1}y$$

Розв'язок методом гауса можна починати не очікуючи на завершення LU факторизації. Використання QUARK без синхронізації між факторизацією і розв'язком методом Гауса може значно пришвидшити обчислення[4].

2.3 Мозайкові матриці

Поширеним методом зберігання матриць є по рядках та по стовпцях. При використанні блочних алгоритмів існують більш ефективні шляхи зберігання матриць.

Використання класичних методів зберігання матриць по рядках у достатньо великих матрицях призводить дого, що елементи одного й того самого блоку можуть опинитися у окремих рядках кешу та навіть у окремих сторінках пам'яті. Проект PLASMA використовує так звані мозайкові матриці. Такі матриці зберігаються як послідовність блоків матриць. Кожен блок вже зберігається по рядках або стовпцях. Це гарантує, що

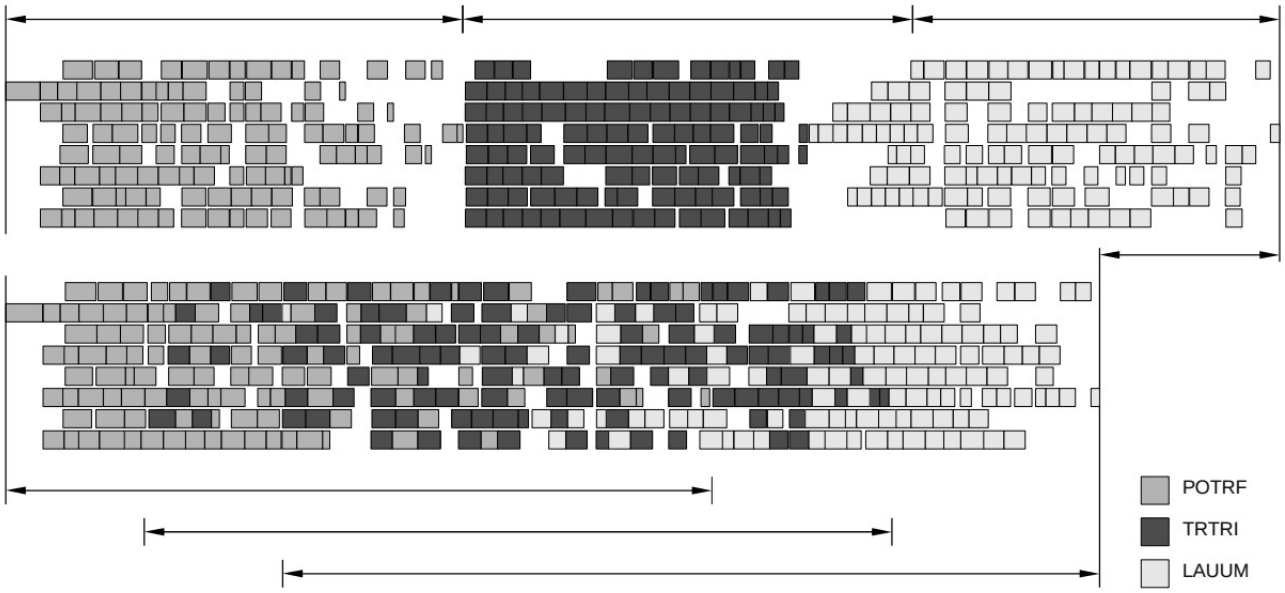


Рис. 6: Порівняння синхронного та асинхронного виконання факторизації Холецького та розв'язку методом Гауса у QUARK[4]

кожен блок буде займати безперервну ділянку пам'яті. Таким чином поліпшується локальність даних, зменшується кількість кеш- та TLB промахів та поліпшується загальна продуктивність обчислень[6].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Рис. 7: Приклад класичного зберігання матриць по рядках

0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51
4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55
8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59
12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63
64	65	66	67	80	81	82	83	96	97	98	99	112	113	114	115
68	69	70	71	84	85	86	87	100	101	102	103	116	117	118	119
72	73	74	75	88	89	90	91	104	105	106	107	120	121	122	123
76	77	78	79	92	93	94	95	108	109	110	111	124	125	126	127
128	129	130	131	144	145	146	147	160	161	162	163	176	177	178	179
132	133	134	135	148	149	150	151	164	165	166	167	180	181	182	183
136	137	138	139	152	153	154	155	168	169	170	171	184	185	186	187
140	141	142	143	156	157	158	159	172	173	174	175	188	189	190	191
192	193	194	195	208	209	210	211	224	225	226	227	240	241	242	243
196	197	198	199	212	213	214	215	228	229	230	231	244	245	246	247
200	201	202	203	216	217	218	219	232	233	234	235	248	249	250	251
204	205	206	207	220	221	222	223	236	237	238	239	252	253	254	255

Рис. 8: Приклад мозайкової матриці

2.4 Функціональне програмування та паралелізм даних

Функціональне програмування, як підмножина декларативного програмування, має багато перевах над імперативним програмуванням. У функціональному програмуванні виконання програми представлено як обчислення чистих функцій. Чистими функціями називаються такі функціх, що не мають побічних ефектів. Функції не можуть зберігати у собі стану, їх значення залежить лише від даних.

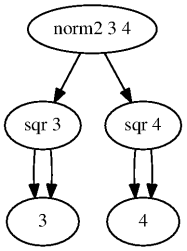
Багато функціональних мов використовують виконання програм через ліниве виконання. Процес виконання програми полягає у тому, щоб виразити значення функції через операції над іншими функціями. Під час виконання права частина відповідної формули замінюється правими частинами функцій, що входять до неї. У процесі таких підстановок будується орієнтований граф, кожна вершина якого представляє собою значення функції. Такий граф (directed acyclic graph, DAG) будується доти, доки усі функції не розгорнуться у операції над константами.

Після побудови DAG виконується так звана редукція графа. Вона полягає у тому, що ті вершини графа, що представляють собою операції над константами заміняються на результат операції над сими константами. У процесі згортання ми отримуємо граф, що складається з однієї вершини, що є результатом обчисленя заданої функції.

Програми, що написані у функціональному стилі не зберігають інформацію про порядок, у якому їхні частини мають бути виконані, тобто

```
sqr x = x * x
norm2 x y = sqr x + sqr y
```

(a)



(b)

Рис. 9: Приклад програми мовою Haskell(a) та відповідний до неї DAG(b)

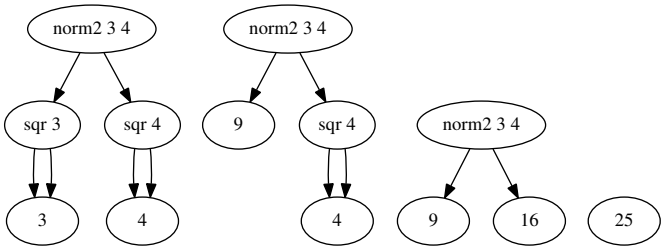


Рис. 10: Приклад редукції графа

вони володіють неявним паралелізмом.

Автоматичне розпаралелювання програм, що написані у функціональному стилі під час компілювання та виконання є предметом багатьох наукових досліджень [7, 8, 9, 10]. Якщо точний розмір вхідних даних не відомий на час компілювання, підчас виконання має бути побудований DAG. Не зважаючі на те, що існує багато методів оптимізації[7], побудова та розпаралелювання DAG може вимагати багато пам'яті та обчислень. Застосовно до лінійної алгебри додаткове навантаження може бути зменшено ща рахунок збільшення мінімального розміру блоку, операції над яким представляються однією вершиною DAG. Більшість блочних алгоритмів лінійної алгебри мають просте представлення у функціональному стилі.

Функціональне програмування також позбуває програміста явного керування пам'ятю. Замість нього виділення та звільнення пам'яти перебирає на себе компілятор або інтерпритатор. Оскільки у функціональному програмуванні кожна операція створює нові данні та жодна зі змінних

не може бути перезаписана, кількість залежностей по даних може бути зменшена, оскільки єдиний тип залежностей, що виникає, це RAW залежність. Таким чином можливий обмін кількості пам'яті, що використовується на кількість залежностей по даних. Цей обмін добре відомий розробникам суперскалярних процесорів, де не зважаючі на залежності по даних деякі операції можливо виконувати одночасно, використовуючі так зване переіменування регістрів. Дослідження показують, що така оптимізація здатна значно прискорити обчислення за рахунок збільшення незалежних вершин у DAG.

Оскільки кількість проміжних обчислень може бути суттєвою, може знадобитись значна кількість додаткової пам'яті для того, щоб їх зберігати. Зазвичай дані, що не є кінцевим результатом обчислень звільняють одразу після того як вони не є у списку залежностей інших даних. Це призводить до того, що програми у функціональному стилі можуть вимагати навіть менше пам'яті, ніж ти, що написані у імперативному, маючі при цьому менше залежностей по даних.

Також функціональне програмування дозволяє уникнути синхронізації між відмінними операціями, такими як декомпозиція матриці та розв'язок системи лінійних рівнянь методом Гауса. Раніше було показано, що така синхронізація значно зменшує продуктивність PLASMA. При цьому функціональне програмування не вимагає явного створення задач та вказування залежностей по даних, як це зроблено у QUARK. Програміст може присвятити усю свою увагу опису алгоритму, дозволяючі оточенню виконання будувати, розпаралелювати та обчислювати DAG.

3 Висновки

У даній роботі були розглянуті існуючі методи розпаралелювання задач лінійної алгебри, запропоновано новий підхід, а саме модифікація DAG паралелізму, при якому використовується не паралелізм задач, а паралелізм даних. Аргументовано наведені переваги запропонованого підходу.

4 Література

[1] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. Communication-Avoiding Parallel Strassen: Implementation and Performance.

[2] Ernie W. Chan. Application of Dependence Analysis and Runtime Data Flow Graph Scheduling to Matrix Computations.

[3] University of Tennessee. PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Archtectures, Version 2.0, November 2009.

[4] Asim YarKhan. Dynamic Task Execution on Shared and Distributed Memory Architectures.

[5] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA.

[6] Neungsoo Park, Bo Hong, and Viktor K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance.

[7] Hans Wolfgang Loidl. Granularity in Large-Scale Parallel Functional Programming.

[8] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, S.L. Peyton Jones. GUM: a portable parallel implementation of Haskell.

[9] Patrick Maier and Phil Trinder. Implementing a High-Level Distributed-Memory Parallel Haskell in Haskell.

[10] F. Warren Burton. Speculative Computation, Parallelism, and Functional Programming.

[11] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, Wei Zhou. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm.

[12] Robert A. van de Geijn, Jerrell Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm.