Паралелно обхождане на граф

Иван Лучев, 62135, Софтуерно Инженерство 3ти курс

Кодът използван в този документ може да бъде достъпен на github.com/luchev

Резултатите от тестовете и диаграмите могат да бъдат достъпени в github или google sheets

Паралелно обхождане на граф
- Анализ на проблема
  - Декомпозиция на проблема
  - Гранулярност
  - Сложност на операциите
  - Многонишкова архитектура на Вариант 1: Генериране на граф
  - Многонишкова архитектура на Вариант 2: Прочитане на граф от файл
- Архитектура на отделните Операции
  - Архитектура на операциите със статично балансиране
  - Архитектура на операциите с динамично балансиране
- Хардуер за извършване на тестовете
- Избор на технологията
  - Описание на алгоритъма за анализ на технологията
  - Реализация на Java
  - Реализация на C++
  - Реализация на Go
  - Заключение
- Операция: Обхождане на граф
  - Serial Breadth first traversal
  - Breadth first traversal with Level Barrier
  - Shallow Traversal
- Операция: Генериране на граф
  - Генериране на насочен граф
  - Генериране на ненасочен граф
- Операция: Прочитане на граф от файл
- Операция: Записване на граф във файл
- Използване на програмата
- Бъдещи планове
- Източници

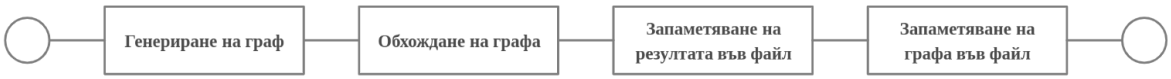
В този документ ще разгледаме обхождане на граф $G(V, E)$ като крайната цел е да получим ефективен алгоритъм, който ни връща като резултат как изглежда функцията на бащите $p(v)$ за някое от покриващите дървета на този граф. Ще разгледаме няколко паралелни алгоритъма, какво ускорение ни предлагат пред последователните обхождания и колко добре скалират с увеличаване на броя на нишките с които работим.

Добре е да уточним, че се фокусираме върху графи с много ребра (**dense graphs**) и за представянето ще използваме **матрица на съседство**.

Анализ на проблема

Задачата е зададена с два варианта на изпълнение.

Вариант 1: Генериране на граф, обхождане на графа и записване на резултата и графа във файлове.



Вариант 2: Прочитане на граф от файл, обхождане на графа и записване на резултата във файл.

Декомпозиция на проблема

От горните две диаграми на изпълнение на програмата може да видим, че имаме архитектура **Pipes & Filters**. Тъй като всяка следваща стъпка в програмата разчита на резултата от предишната, т.е имаме **зависимост по данни** (пр. Обхождането на граф предполага че разполагаме с граф, било то прочетен от файл или генериран). При такава архитектура на цялото приложение, за да постигнем максимално бързо изпълнение на многопроцесорна машина е необходимо да паралелизираме всяка една независима стъпка на програмата. Всяка операция, разделя данните с които работи за да може да ги обработи паралелно. Тоест имаме **SPMD** декомпозиция.

Гранулярност

Ще използваме **средна** грануларност от порядъка на 1 връх (т.е. 1 ред от матрицата на съседство). Избираме тази гранулярност, за да може да се възползваме от кеша на процесора и да получим максимално бързодействие на програмата. При по-финна гранулярност, кеша на процесора няма да бъде използван достатъчно оптимално и кодирането на заданията няма да е толкова просто, което ще доведе до забавяне в комуникацията. По-едра гранулярност няма да доведе до добро ниво на паралелизация на алгоритмите, поради факта че една нишка ще трябва да върши работата на няколко нишки при текущата избрана средна гранулярност.

За доказателство на тези твърдения може да разгледаме времето, за което програмата се изпълнява при генериране на насочен (използва кеша на процесора) и ненасочен граф (не използва кеша достатъчно).

Сложност на операциите

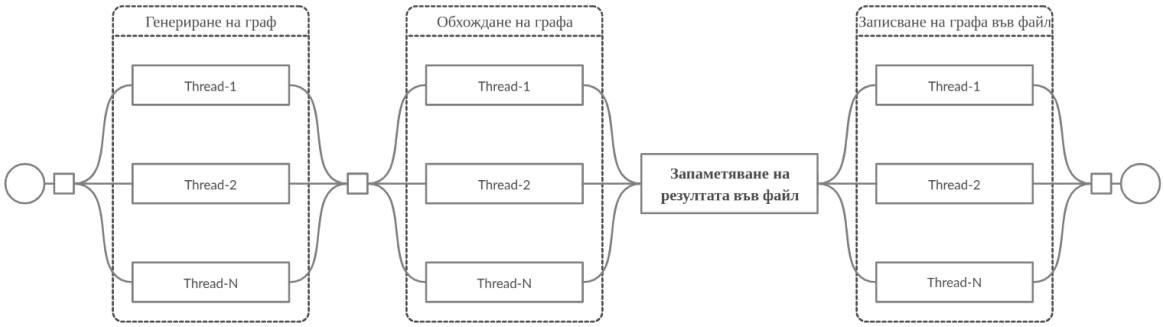
За да преценим кои операции е удачно да бъдат паралелизирани трябва да оценим сложнсоттан всяка една от тях. Като имаме в предвид че разглеждаме плътни графи, т.е $|E| \to |V|^2$

Операция	Сложност
Генериране на граф $G(V, E)$	$\Theta(\|E\|) = \Theta(\|V\|^2)$
Прочитане на граф от файл $G(V, E)$	$\Theta(\|E\|) = \Theta(\|V\|^2)$
Обхождане на граф $G(V, E)$	$\mathcal{O}(\|E\|) = \mathcal{O}(\|V\|^2)$
Запаметяване на функцията на бащите $p(v)$ за граф $G(V, E)$	$\Theta(\|V\|)$
Запаметяване на граф $G(V, E)$ във файл	$\Theta(\|E\|) = \Theta(\|V\|^2)$

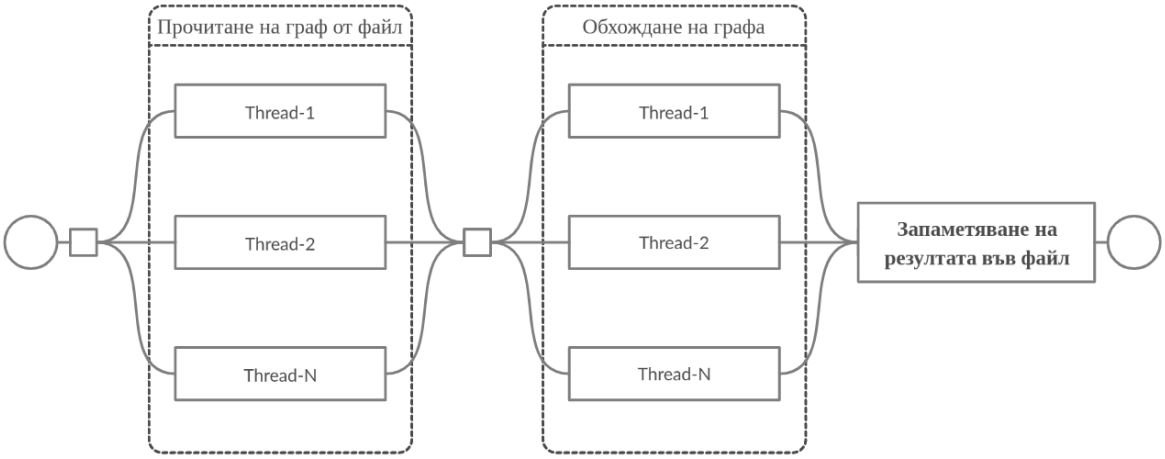
Запаметяване на функцията на бащите е най-бързата операция - единствената такава, която скалира линейно спрямо броя на върховете. Всички останали операции бавни - генериране, прочитане, обхождане и запаметяване на граф скалират квадратично спрямо броя на върховете на графа. За да постигнем максимално ускорение чрез използването на многонишков алгоритъм ще се фокусираме върху паралелизирането на бавните операции.

На следните 2 графики е представена архитектурата с операциите, които целим да направим многонишкови, за да ускорим тяхната обработка.

Многонишкова архитектура на Вариант 1: Генериране на граф



Многонишкова архитектура на Вариант 2: Прочитане на граф от файл

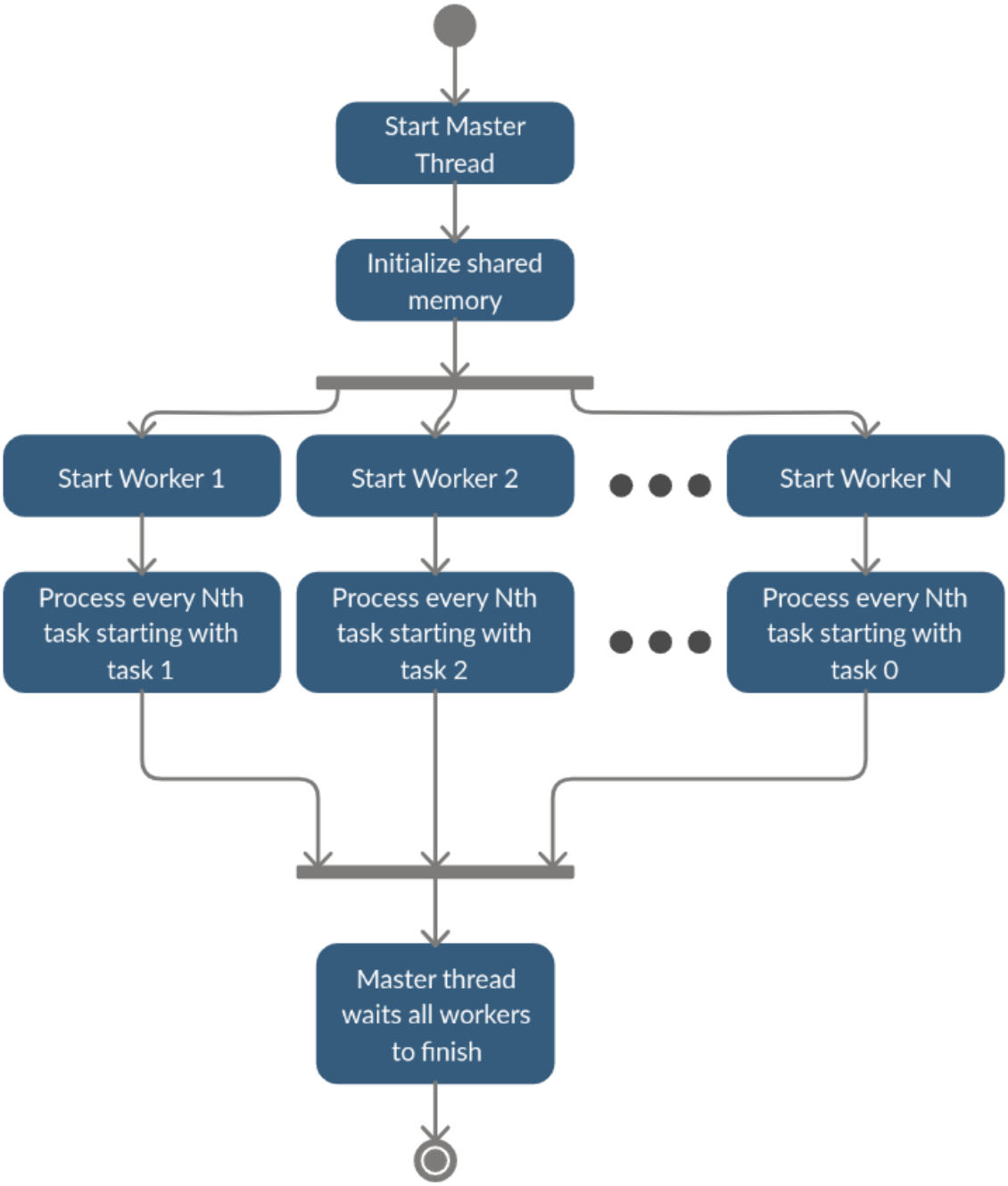


Архитектура на отделните Операции

Ще използваме 2 вида разпределение на задачите - динамично и статично. Където използваме **динамично разпределение** ще използваме **Master-slaves** архитектура. За **статично балансиране** ще използваме **споделена памет**.

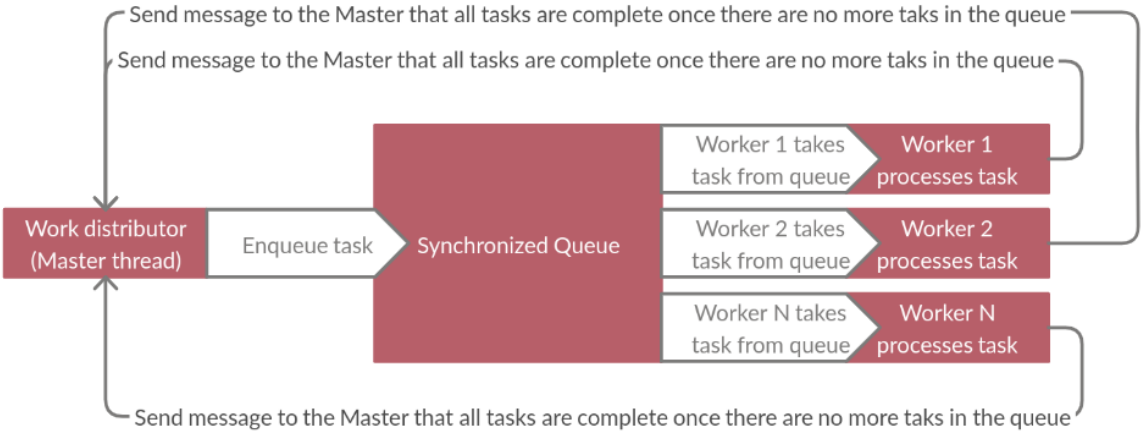
Архитектура на операциите със статично балансиране

При статично балансиране можем предварително да определим всеки един worker точно кои задачи трябва да изпълни, за това не е необходима синхронизация между тях, тъй като всеки worker работи независимо от останалите. За да не прехвърляме памет между отделните нишки използваме споделена памет за максимално бързодействие. Единствената синхронизация, която трябва да направим е основната нишка да изчака всички останали да приключат работата си - това може да направим с прост семафор.



Архитектура на операциите с динамично балансиране

При операциите с динамично балансиране ще използваме Master-slaves архитектура, като Master процеса се грижи за стартирането на worker процесите и задаването на задачите им. За да избегнем голямо натоварване от постоянно изпращане на съобщения между Master и worker нишките, използваме една синхронизирана опашка, където Master процеса само подава задачи за обработване, а worker нишките взимат задачи от тази опашка когато приключат обработването на предишната си задача. Синхронизираната опашка може да се окаже тясно място в системата, но както ще видим от резултатите от тестовете при графи с много върхове (много повече от процесорите с който разполагаме), не оказва влияние на ускорението.



Хардуер за извършване на тестовете

Сравнението на алгоритмите, тяхната ефективност, възможност да бъдат паралелизирани и колко добре скалират с увеличаване на броя на процесорите ще тестваме на Linux машина t5600. Тук са показани най-важните характеристики на тази машина, които ще имат значение за резултатите и изводите, които правим.

Операционната система на машината е Linux 3.10. Необходимият софтуер е единствено компилатор на C++, Java и Go.

16-Core

```
CPU(s):              32
Thread(s) per core:  2
Core(s) per socket:  8
Socket(s):           2
CPU MHz:             1880.017
L1d cache:           32K
RAM:                 64G
```

Избор на технологията

Преди да разгледаме и анализираме алгоритми за всяка една операция, описана в горната точка, ще изберем подходяща технология за тестване.

Ще разгледаме Java, C++ и Go. Ще тестваме предимствата и недостатъците на всяка една технология като използваме един и същ алгоритъм за генериране на насочен граф.

Всеки тест е повторен по 3 пъти на машината **16-Core**. Но тъй като избирането на технология не е фокус на този документ, ще разгледаме само най-добрите резултати от тестването на базата на които ще направим изводи.

Описание на алгоритъма за анализ на технологията

Генерираме насочен граф. Използваме **статично балансиране** - разпределяме върховете на графа на броя на нишките с които работим. Всяка нишка обработва еднакъв брой върха на принципа на **Round-Robin**. По този начин всяка нишка генерира ребрата на равен брой върхове.

Пример: ако имаме 16 нишки, то нишка 1 обработва върхове, чиито индекси дават модул 1 при деление на 16 (върхове 1, 17, 33, ...), нишка 3 обработва върховете, чиито индекси дават модул 3 при деление на 16 (върхове 3, 19, 35, ...).

Реализация на Java

Резултат от тестването

Math.random	Threads	Vertices	Time (seconds)	Speedup
	1	10,000	3.0796	1
	4	10,000	26.1181	0.11

ThreadLocalRandom	Threads	Vertices	Time (seconds)	Speedup
	1	40,000	10.3472	1
	4	40,000	3.4738	2.97
	8	40,000	2.3720	4.36
	16	40,000	2.0972	4.93
	32	40,000	1.9776	5.23

Извод

Java има два вградени метода за генериране на случайни числа.

Math.random - изключително бавен метод за генериране на случайни числа, който се забавя 10х когато увеличим броят на нишките от 1 на 4. Повече тестове не са необходими. Math.random е неблагоприятен за паралелно обработване.

ThreadLocalRandom - бърз алгоритъм за генериране на случайни числа, който работи добре с повече нишки. За съжаление ускорението не се доближава до оптималното линейно като при 16 нишки достигаме 5х ускорение, при оптимално 16х. ThreadLocalRandom е значително по-добър от Math.random, но не е достатъчно добър за оптимални резултати.

> Java не е подходяща технология, защото не предоставя достатъчно добро ускорение с увеличаване на броя на нишките на които работим

Реализация на C++

Резултати от тестването

Mersenne Twister	Threads	Vertices	Time (seconds)	Speedup
Non-Threaded	1	10,000	32 минути	1
Non-Threaded	16	10,000	43 минути	0.74
Threaded	1	10,000	6.8707	1
Threaded	4	10,000	1.7845	3.85
Threaded	8	10,000	0.9108	7.54
Threaded	16	10,000	0.5386	12.75
Threaded	32	10,000	0.4383	15.67

Marsaglia's xorshf	Threads	Vertices	Time (seconds)	Speedup
Non-Threaded	1	20,000	3.6333	1
Non-Threaded	4	20,000	10.3713	0.35
Threaded	1	20,000	3.4334	1
Threaded	4	20,000	0.9047	3.7
Threaded	8	20,000	0.4735	7.2
Threaded	16	20,000	0.3204	10.71
Threaded	32	20,000	0.2857	12

Algorithm (threaded)	Threads	Vertices	Time (seconds)	Speedup
Mersenne Twister	1	50,000	172.0770	1
Mersenne Twister	16	50,000	12.0559	14.27
Mersenne Twister	32	50,000	9.51459	18
Marsaglia's xorshf	1	50,000	21.4722	1
Marsaglia's xorshf	16	50,000	1.6168	13.28
Marsaglia's xorshf	32	50,000	1.6420	13

Извод

Разглеждаме вграденият в C++ Mersenne Twister алгоритъм и имплементация на Marsaglia's xorshf.

Mersenne Twister работи сравнително бавно, за това тестваме с по-малък брой върхове. Въпреки това алгоритъма предоставя много добро ускорение в сравнение с Java, като при 16 нишки получаваме ускорение 12.17x.

Marsaglia's xorshf е в пъти по-бърз от Mersenne Twister. Поради тази причина увеличаваме броят на върховете на 20,000, за да може да оценим ускорението. Алгоритъмът не скалира толкова добре спрямо броят нишки като при 16 нишки получаваме 10.71x ускорение, което е 2 пъти по-добро от това на Java, но е значително по-лошо от ускорението на Mersenne

Twister.

> За голям брой върхове въпреки че ускорението на Marsaglia's xorshf е по-лошо от това на Mersenne Twister, Marsaglia's xorshf работи по-бързо и би бил-по-добрият избор за малък брой ядра (16).

Реализация на Go

Резултати от тестовете

Cryptographic pseudo-random number sequence generator (CPRNG)	Threads	Vertices	Time (seconds)	Speedup
	1	40,000	12.8282	1
	4	40,000	6.3824	2
	8	40,000	5.4181	2.36
	16	40,000	5.7462	2.23
	32	40,000	6.7325	1.9

Pseudo-random number generator (PRNG)	Threads	Vertices	Time (seconds)	Speedup
	1	40,000	33.8937	1
	4	40,000	8.3982	4
	8	40,000	4.3595	7.77
	16	40,000	2.4213	14
	32	40,000	2.1362	15.86

Pseudo-random number sequence generator (PRNG Sequence)	Threads	Vertices	Time (seconds)	Speedup
	1	40,000	4.5132	1
	4	40,000	1.2269	3.67
	8	40,000	0.6593	6.84
	16	40,000	0.3996	11.29
	32	40,000	0.3498	12.9

Извод

Go има 2 начина за генериране на случайни числа - криптографски (недетерминиран) и псевдо-случаен генератор (детерминиран).

Cryptographic pseudo-random number sequence generator (**CPRNG**) генерира случайни числа по недетерминистичен начин. Това забавя работата му изключително много и получаваме ускорение само 2.23x при 16 нишки.

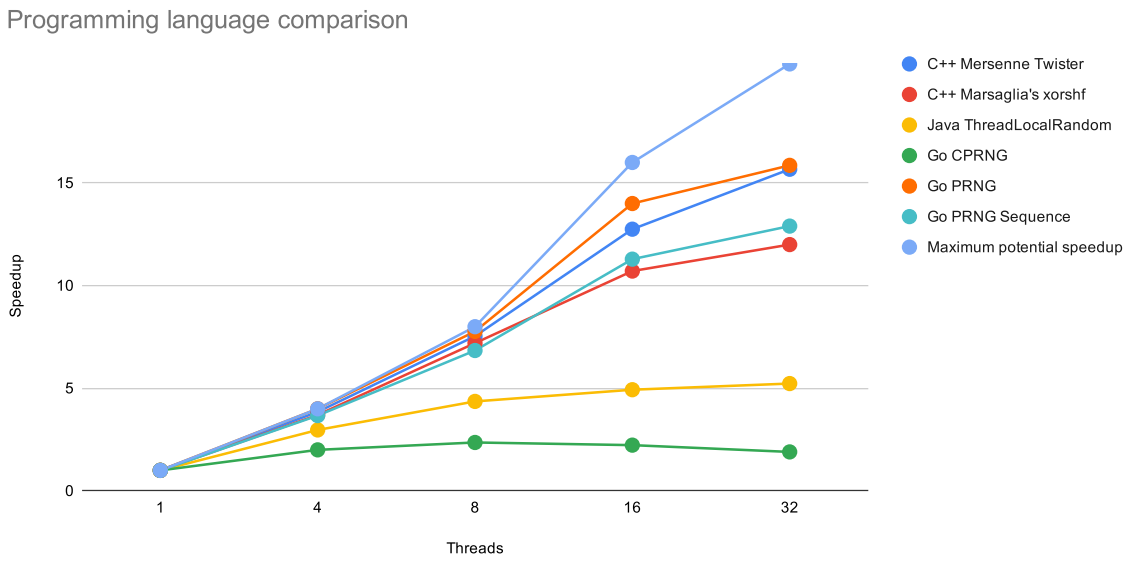
Pseudo-random number generator (**PRNG**) генерира случайни числа по детерминистичен начин. Това позволява много по-добро ускорение с използването на повече нишки - до 14x ускорение.

Ако използваме locality на данните и генерираме поредица от случайни числа (**PRNG Sequence**) получаваме само 11.29x ускорение на 16 нишки, но поради locality получаваме 6 пъти по-добро време.

> За машина с малък брой ядра (16) PRNG Sequence е по-подходящ и работи по-бързо от PRNG, въпреки че не скалира толкова добре с броя нишки. Ако увеличим броят логически ядра на процесора или броят върхове PRNG ще е по-добрият алгоритъм.

Заключение

Threads	1	4	8	16	32
C++ Mersenne Twister	1	3.85	7.54	12.75	15.67
C++ Marsaglia's xorshf	1	3.7	7.2	10.71	12
Java ThreadLocalRandom	1	2.97	4.36	4.93	5.23
Go CPRNG	1	2	2.36	2.23	1.9
Go PRNG	1	4	7.77	14	15.86
Go PRNG Sequence	1	3.67	6.84	11.29	12.9
Linear speedup	1	4	8	16	20.8



Най-добро ускорение на 32 нишки получаваме при Go - 15.86x (PRNG), като C++ (Mersenne Twister) е на второ място с 15.67x. Java е на последно място с 5.23x ускорение на 32 нишки.

За този проект ще изберем Go като технология за многонишкова обработка на данни.

Важно е да забележим, че на машината на която се извършват тестовете има 16 физически ядра на процесора с hyperthreading, което ни показва че имаме 32 логически ядра. Въпреки това дори когато използваме 32 нишки получаваме оптимално ускорение 15.86x за алгоритъма PRNG на Go, което се доближава до броят на физическите 16 ядра, но по

никакъв начин не се доближава до логическите 32 ядра. Това означава, че може да очакваме ускорение на програмата в зависимост от това колко физически ядра има процесора, не колко логически.

Операция: Обхождане на граф

Serial Breadth first traversal

За сравнение с останалите алгоритми ще използваме стандартна имплементация на обхождане в ширина, която използва само една нишка. Обработването на данните е последователно. Предимството на тази имплементация е че няма забавяне от страна на синхронизация. Недостатъкът е, че не може да скалира с увеличаване на броят на ядрата процесора на системата.

Breadth first traversal with Level Barrier

Идея на алгоритъма

Ще разгледаме имплементация на BFS с level barrier, като паралелен алгоритъм за обхождане на граф. Алгоритъмът се базира на стандартното обхождане в ширина. Паралелизмът при този алгоритъм идва от факта че ако сме обходили ниво N на графа, имаме множество от върховете от ниво $N+1$. Всеки един от тези върхове от ниво $N+1$ от графа може да бъде обходен паралелно и може да се открият съседите му независимо от откриването на съседите на всички останали върхове от ниво $N+1$. Единствената особеност е, че трябва попълването на множеството на върхове от ниво $N+2$ от графа да е синхронизиран процес. За да се справим със синхронизацията ще имплементираме алгоритъма с **динамично балансиране** и архитектура **Master-slaves**. Динамично балансиране е необходимо защото не знаем точно колко върха ще има на ниво N от графа и не може да определим константен брой върхове, които всяка нишка да обработва. Архитектурата е от тип Master-slaves, за да може лесно да се справим с проблема на синхронизацията при попълване на множеството на върхове от ниво N на графа.

Комуникация и разпределение на работата между отделните нишки

Основната нишка на алгоритъма се грижи за създаването на задачи в синхронизирана опашка, които да бъдат раздадени на предварително стартирани worker-нишки. Тези worker-нишки чакат да получат задача (връх от графа, който да обработят) от опашката, след което намират всичките съседи на разстояние 1 и ако има съсед, който не е посетен до този момент изпращат съобщение към основната нишка с неговия номер. Основната нишка проверява дали този връх не е бил вече добавен към множеството от върховете от следващото ниво на графа и ако не - го добавя.

Защо централизирана архитектура не е проблем за обхождане на граф.

Основната нишка може да бъде видяна като тясно място за алгоритъма и потенциално да лимитира неговото бързодействие, тъй като тя играе ролята на Master и синхронизира работата между всички останали нишки. Този проблем ще съществува само когато броят на върховете на графа е близък до броят на ядрата с които разполага процесора на машината - това са незначително малки графи, дори да имаме 1000 процесора. В другите случаи (когато броят на върховете на графа са много повече от ядрата на процесора) всеки един worker ще трябва да извършва доста продължителна работа и рядко да изпраща съобщения към основната нишка. Допълнително, за бързодействие и намаляване на броят на синхронизираните съобщения, които трябва да се изпращат - използваме споделена памет, за да следим кои върхове вече са обходени и добавени. Преди някой worker да изпрати синхронизирано съобщение до основната нишка, че иска да добави нов

връх за следващото ниво от графа се допитва до тази споделена памет - дали вече е бил добавен този съсед или не. Тази споделена памет не е защитена с mutex, което води до бърз достъп до нея, но и означава че може да настъпи race condition. Това не е проблем защото в случай на race condition ще бъдат изпратени повече от 1 синхронизирани съобщения към основната нишка с един и същи връх, но основната нишка проверява (строго, за разлика от споделената памет) дали даден връх е бил вече добавен за следващото ниво и съответно ще игнорира второто съобщение за добавяне.

Възможен ли е алгоритъм с нецентрализирана архитектура и ще подобри ли скоростта?

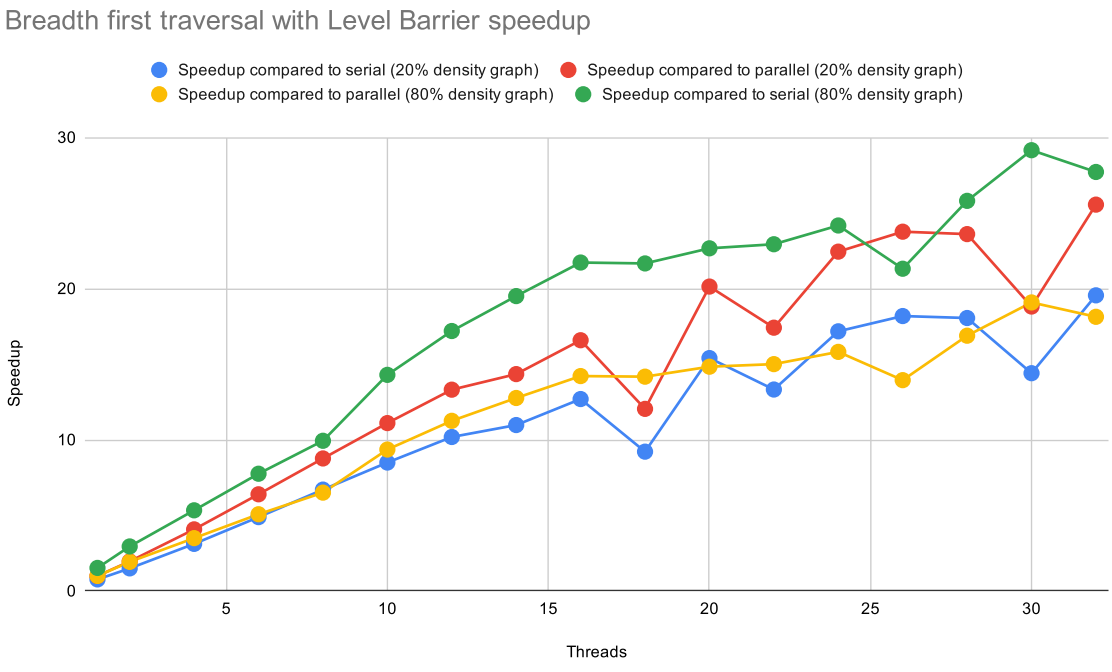
Възможно е да се реализира алгоритъм подобен на гореописания, но с нецентрализирана архитектура и разпределена памет. Тоест, няма да има една Master нишка, която да определя задачите на всички останали и да синхронизира съобщенията между тях. Заданията може да се разпределят пак по подобен начин - със синхронизирана опашка, но за да се синхронизира обмена на информация между отделните нишки ще трябва да се използва обмен на съобщения тип всеки-към-всеки. Това ще доведе до голям overhead на работата, която всяка една нишка трябва да свърши, заради множеството съобщения, които трябва да обработва. Например ако използваме 64 нишки и нишка 24 иска да добави нов връх от следващото ниво на обхождане, трябва да се допита до всички други 64 нишки дали те вече не са го добавили и чак тогава да го добави.

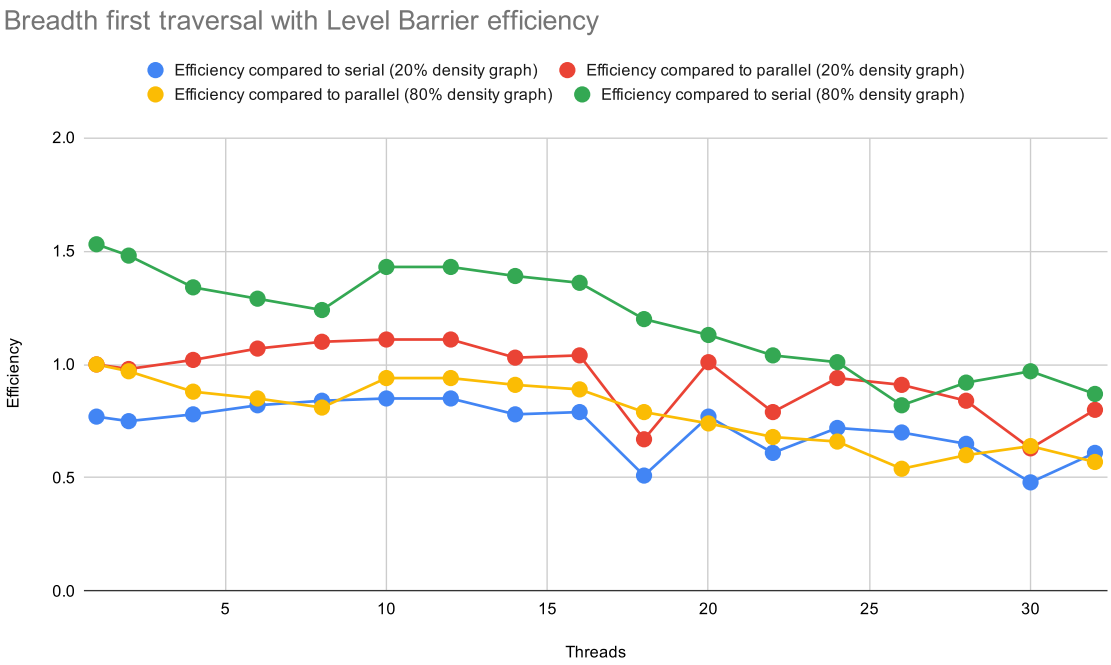
Breadth first traversal with Level Barrier върху граф с 50,000 върха и 20% density

Threads	T1	T2	T3	Tp = min()	Speedup(compared to parallel)	Efficiency (compared to parallel)	Speedup (compared to serial)	Efficiency (compared to serial)
1	13653	13491	13290	13290	1	1	0.77	0.77
2	6765	6789	6896	6765	1.96	0.98	1.5	0.75
4	3615	3260	3364	3260	4.08	1.02	3.12	0.78
6	2162	2078	2584	2078	6.4	1.07	4.89	0.82
8	1613	1515	1812	1515	8.77	1.1	6.71	0.84
10	1263	1196	1601	1196	11.11	1.11	8.5	0.85
12	1352	1091	998	998	13.32	1.11	10.19	0.85
14	945	1111	926	926	14.35	1.03	10.98	0.78
16	840	801	879	801	16.59	1.04	12.7	0.79
18	1103	1402	1288	1103	12.05	0.67	9.22	0.51
20	660	744	1093	660	20.14	1.01	15.41	0.77
22	991	763	1000	763	17.42	0.79	13.33	0.61
24	1153	638	592	592	22.45	0.94	17.18	0.72
26	628	1101	559	559	23.77	0.91	18.19	0.7
28	563	852	606	563	23.61	0.84	18.06	0.65
30	706	747	902	706	18.82	0.63	14.41	0.48
32	520	707	610	520	25.56	0.8	19.56	0.61

Breadth first traversal with Level Barrier върху граф с 50,000 върха и 80% density

Threads	T1	T2	T3	Tp = min()	Speedup(compared to parallel)	Efficiency (compared to parallel)	Speedup (compared to serial)	Efficiency (compared to serial)
1	11077	10268	11973	10268	1	1	1.53	1.53
2	5429	6198	5310	5310	1.93	0.97	2.95	1.48
4	2937	3055	3074	2937	3.5	0.88	5.34	1.34
6	2152	2022	2263	2022	5.08	0.85	7.76	1.29
8	1622	1578	1725	1578	6.51	0.81	9.94	1.24
10	1097	1454	1185	1097	9.36	0.94	14.3	1.43
12	912	1259	939	912	11.26	0.94	17.2	1.43
14	804	845	811	804	12.77	0.91	19.51	1.39
16	761	722	783	722	14.22	0.89	21.73	1.36
18	731	724	778	724	14.18	0.79	21.67	1.2
20	704	692	697	692	14.84	0.74	22.67	1.13
22	1036	1155	684	684	15.01	0.68	22.94	1.04
24	649	1064	756	649	15.82	0.66	24.18	1.01
26	850	762	736	736	13.95	0.54	21.32	0.82
28	816	1025	608	608	16.89	0.6	25.81	0.92
30	557	538	545	538	19.09	0.64	29.16	0.97
32	586	566	1080	566	18.14	0.57	27.72	0.87





Извод

Въпреки че използваме динамично балансиране, алгоритъмът се представя много добре с ускорение от порядъка на 20 пъти при наличие на 16 процесора. При графи с 20% density на ребрата получаваме на места ускорение от порядъка на 25-30 пъти, което е нереалистично и се получава в следствие от небалансирани графи с малък брой ребра (защото генерираме случаен граф за изпълнение на тестовете). Реална оценка за ускорението може да видим при графи с 80% density на ребрата - получаваме ускорение от порядъка на 16-18 пъти спрямо паралелният алгоритъм и около 25 пъти спрямо стандартната реализация на BFS. Толкова добро ускорение в сравнение със стандартната реализация на BFS се получава поради забавянето при зареждане на данните от рам паметта в процесора, което до някаква степен се паралелизира при Breadth first traversal with Level Barrier алгоритъма. Добре е да отбележим и че при графи с малък брой ребра ускорението на паралелният алгоритъм спрямо себе си е по-добр от колкото ускорението спрямо последователният алгоритъм, но при графи с голям брой ребра имаме обратният ефект - ускорението на паралелният алгоритъм спрямо последователният алгоритъм е по-добро. Това е добър знак за алгоритъма - скалира добре при по-голямо количество данни.

Shallow Traversal

Подобряване на времето за обхождане като загубваме идеята за ниво на графа

Breadth first traversal with Level barrier има едно тясно място и това е, че на всяко ниво трябва да спре и да изчака всички нишки да обработят върховете от това ниво преди да почне да обработва следващото ниво. Тук ще предложа алгоритъм, който ще обходи графа, без да има нужда от синхронизация на всяко ниво от графа, но ще изгуби информацията за това на какво ниво се намираме.

Идея на алгоритъма

Вместо да пускаме BFS от 1 връх, може да го пуснем от няколко върха едновременно на отделни нишки. Въпросът е как знаем кога да спрем една нишка защото работи твърде дълго, а другите нишки са приключили работа. Ще фиксираме колко нива да обхожда всяка една нишка релативно спрямо върха от който е започнала обхождането. Възможно е да го фиксираме на произволна константа, но за целите на този документ ще фиксираме нивата

на обхождане на 1. По този начин всяка една нишка стартира работа по произволен връх и проверява всеки един негов съсед дали има баща. Ако съседният връх няма баща, то текущият връх отбелязваме като негов баща. По този начин като обходим всички върхове ще получим покриващо дърво/гора на графа. Тази идея наподобява алгоритъма на Крускал, като проверява дали може да изгради ребро между два върха и ги свързва. За такъв алгоритъм може да използваме **статично балансиране**, защото работата може предварително да я разпределим по равно на всяка нишка.

Недостатъци

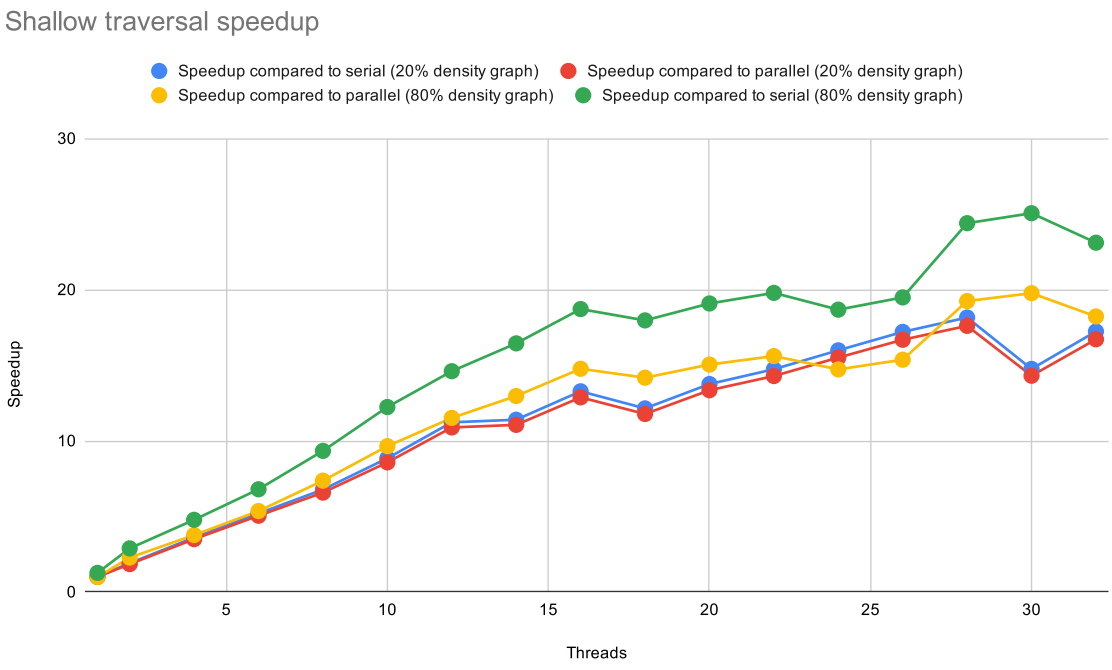
Поради факта, че пускаме обхождането от различни върхове, а не от 1, губим информацията за ниво на връх в графа, тъй като това ниво е релативно спрямо случайният начален връх. Ако не ни трябва да знаем нивото на върховете, това е добра замяна - повече скорост, за малко загуба на информация.

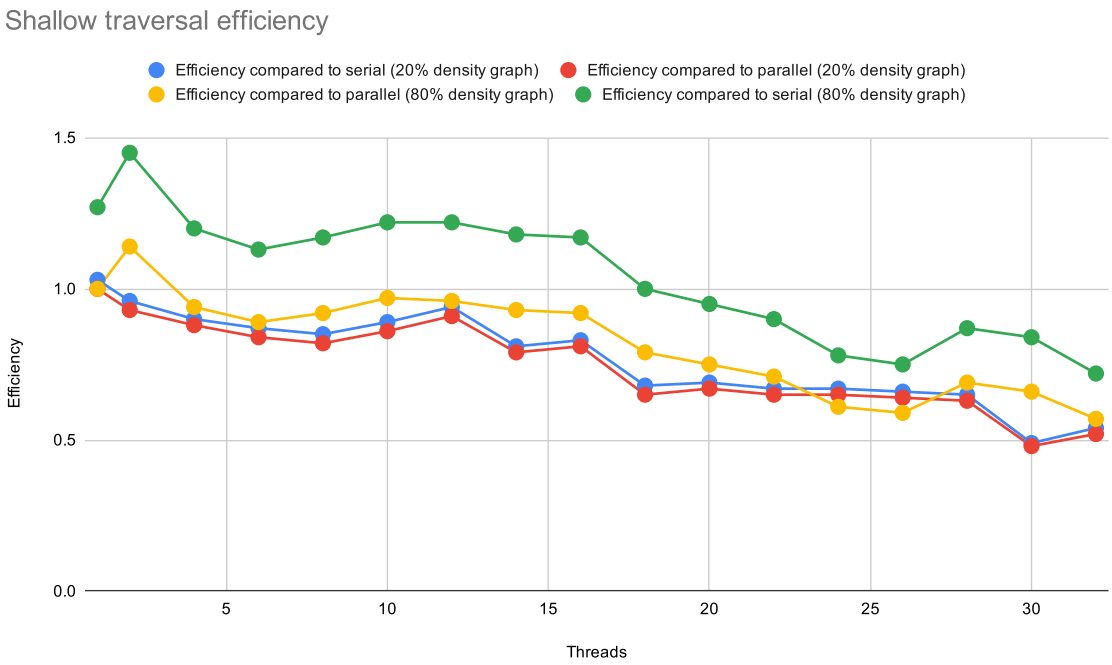
Shallow Traversal върху граф с 50,000 върха и 20% density

Threads	T1	T2	T3	Tp = min()	Speedup(compared to parallel)	Efficiency (compared to parallel)	Speedup (compared to serial)	Efficiency (compared to serial)
1	10290	9863	12432	9863	1	1	1.03	1.03
2	5337	5937	6059	5337	1.85	0.93	1.91	0.96
4	2844	2814	5421	2814	3.5	0.88	3.61	0.9
6	1954	2105	2124	1954	5.05	0.84	5.2	0.87
8	1564	1555	1499	1499	6.58	0.82	6.78	0.85
10	1149	1308	1356	1149	8.58	0.86	8.85	0.89
12	1219	1046	906	906	10.89	0.91	11.23	0.94
14	892	1111	1236	892	11.06	0.79	11.4	0.81
16	826	766	877	766	12.88	0.81	13.28	0.83
18	837	1391	1191	837	11.78	0.65	12.15	0.68
20	808	749	739	739	13.35	0.67	13.76	0.69
22	690	819	724	690	14.29	0.65	14.74	0.67
24	636	657	772	636	15.51	0.65	15.99	0.67
26	591	636	705	591	16.69	0.64	17.21	0.66
28	560	568	620	560	17.61	0.63	18.16	0.65
30	703	689	757	689	14.31	0.48	14.76	0.49
32	727	590	697	590	16.72	0.52	17.24	0.54

Shallow Traversal върху граф с 50,000 върха и 80% density

Threads	T1	T2	T3	Tp = min()	Speedup(compared to parallel)	Efficiency (compared to parallel)	Speedup (compared to serial)	Efficiency (compared to serial)
1	12604	12375	12861	12375	1	1	1.27	1.27
2	5429	6198	6306	5429	2.28	1.14	2.89	1.45
4	3486	3506	3279	3279	3.77	0.94	4.78	1.2
6	2359	2505	2307	2307	5.36	0.89	6.8	1.13
8	1680	1691	1718	1680	7.37	0.92	9.34	1.17
10	1298	1443	1282	1282	9.65	0.97	12.24	1.22
12	1082	1074	1078	1074	11.52	0.96	14.61	1.22
14	961	954	983	954	12.97	0.93	16.45	1.18
16	838	879	962	838	14.77	0.92	18.72	1.17
18	873	890	911	873	14.18	0.79	17.97	1
20	822	836	840	822	15.05	0.75	19.09	0.95
22	939	810	793	793	15.61	0.71	19.79	0.9
24	840	893	881	840	14.73	0.61	18.68	0.78
26	878	828	805	805	15.37	0.59	19.49	0.75
28	748	670	643	643	19.25	0.69	24.4	0.87
30	668	627	626	626	19.77	0.66	25.06	0.84
32	722	679	705	679	18.23	0.57	23.11	0.72





Извод

Този начин на обхождане на граф не предоставя достатъчно добро ускорение, въпреки че използва статично балансиране. Ускорението което постигаме е 16-18 пъти на машина с 16 процесора, което е задоволително, но значително по-лошо от ускорението на алгоритъма Breadth first traversal with Level Barrier, като се има предвид, че дори имаме загуба на информация. Този алгоритъм е подходящ за изпълнение на машина с голям брой процесори (или върху видео карта), където може да се възползваме от много по-голям паралелизъм, защото няма нужда от комуникация между отделните процесори и всички процесори изпълняват работата си независими.. Но на машина с 16 процесора този алгоритъм не се представя много добре.

Операция: Генериране на граф

Генериране на насочен граф

Алгоритъмът за генериране на насочен граф представен чрез матрица на съседство е обхождане на матрицата по ред (начален връх) и стълб (краен връх) и отбелязване с 1, ако има ребро между тях, и 0 ако няма. За генерирането на 1 или 0 използваме случайно генерирани числа.

Този алгоритъм е подходящ за **статично балансиране**, тъй като предварително може да предвидим точно колко работа има и работата е равно разпределена. Използваме разпределена архитектура със споделена памет. Това е възможно защото нишките работят независимо 1 от друга и няма да възникне race condition. Разпределяме броят на върховете по равно на всяка нишка на принципа на **Round-robin**. Върху един връх (1 ред от матрицата) работи само 1 нишка, за да се възползваме от locality на данните, тъй като данните се пазят във масив.

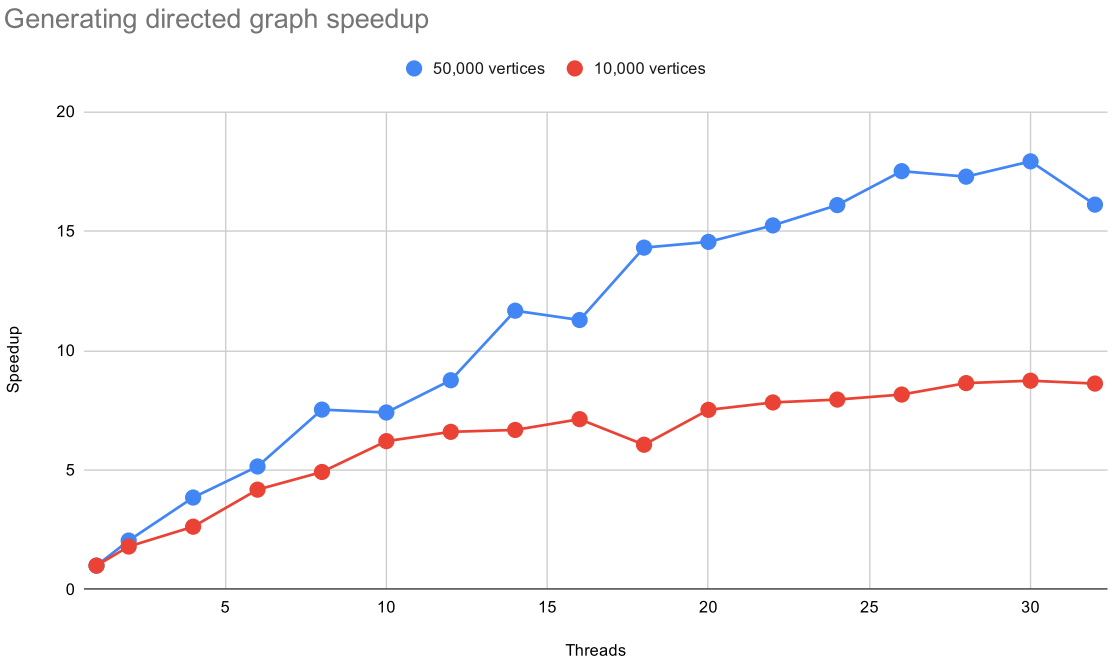
Генериране на насочен граф с 10,000 върха

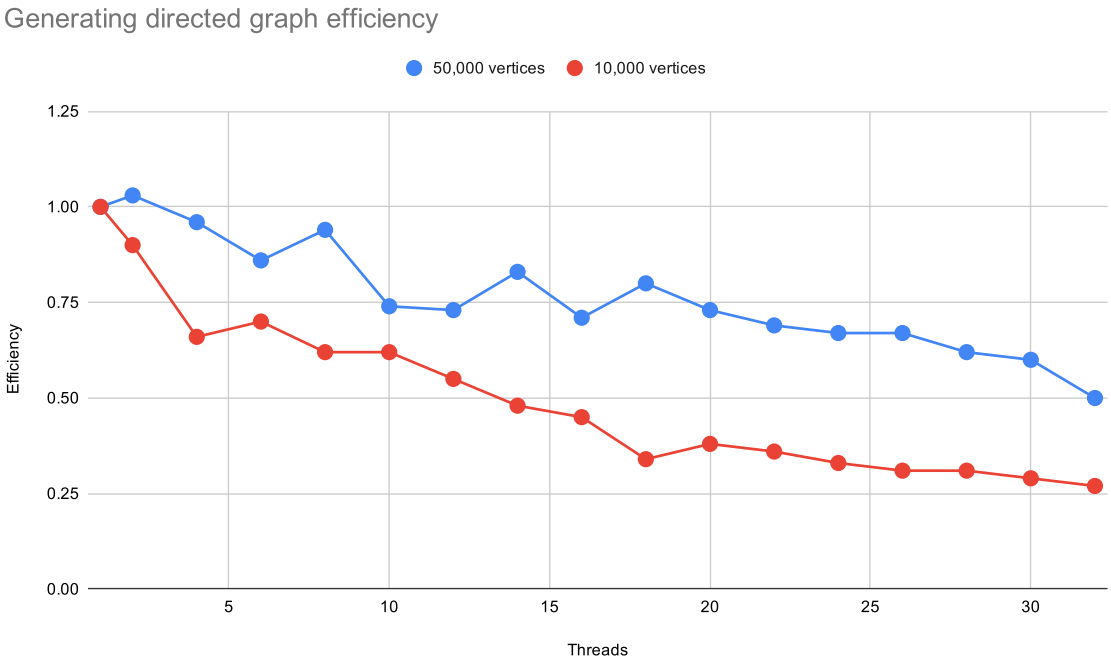
Threads	T1	T2	T3	Tp = min()	Speedup	Efficiency
1	480.699211	421.350663	498.707788	421.35	1	1
2	248.273344	257.322738	233.657925	233.66	1.8	0.9
4	181.678693	203.931826	160.360262	160.36	2.63	0.66
6	122.825486	110.582942	100.816474	100.82	4.18	0.7
8	88.265943	88.806896	85.56125	85.56	4.92	0.62
10	82.130222	67.85724	111.561056	67.86	6.21	0.62
12	71.727303	67.362187	63.866451	63.87	6.6	0.55
14	72.222712	63.097964	64.237934	63.10	6.68	0.48
16	156.154316	63.219007	59.09952	59.10	7.13	0.45
18	131.708571	113.398306	69.541326	69.54	6.06	0.34
20	76.8928	56.000011	61.341353	56.00	7.52	0.38
22	59.400639	64.725446	53.816368	53.82	7.83	0.36
24	52.984928	53.383054	58.604692	52.98	7.95	0.33
26	51.656717	247.857301	57.601629	51.66	8.16	0.31
28	50.39543	52.848971	48.784257	48.78	8.64	0.31
30	58.977379	60.737161	48.186554	48.19	8.74	0.29
32	48.900979	57.939858	52.893174	48.90	8.62	0.27

Генериране на насочен граф с 50,000 върха

Threads	T1	T2	T3	Tp = min()	Speedup	Efficiency
1	15565.12	16429.31	18286.2	15565.12	1	1
2	9568.61	8798.07	7593.73	7593.73	2.05	1.03
4	4043.67	4043.67	5207.05	4043.67	3.85	0.96
6	3819.71	3024.04	3235.78	3024.04	5.15	0.86
8	2076.52	2354.86	2067.25	2067.25	7.53	0.94
10	2108.81	2297.96	2100.91	2100.91	7.41	0.74
12	1777.83	2021.12	1851.91	1777.83	8.76	0.73
14	2642.97	1419.88	1333.73	1333.73	11.67	0.83
16	1409.14	2011.82	1379.31	1379.31	11.28	0.71
18	1523.03	1087.42	1158.73	1087.42	14.31	0.8
20	1110.09	1070.01	1374.19	1070.01	14.55	0.73
22	1097.17	2607.36	1021	1021.00	15.24	0.69

Threads	T1	T2	T3	Tp = min()	Speedup	Efficiency
24	989.1	990.17	967.27	967.27	16.09	0.67
26	1071.06	907.54	888.83	888.83	17.51	0.67
28	1919.83	900.61	1850.97	900.61	17.28	0.62
30	1608.68	868.82	1445.68	868.82	17.92	0.6
32	966	966.71	1001.54	966.00	16.11	0.5





Извод

Генерирането на насочен граф може да се паралелизира много добре. 18 пъти ускорение на 16 ядрен процесор е почти оптимално време. За да постигнем толкова добор ускорение е необходимо да имаме достатъчно голям граф. При малки графи ускорението не може да бъде измерено толкова добре, защото голяма част от времето е отделена за заделяне на

памет, стартиране на всички нишки, context switching и тн.

Генериране на ненасочен граф

Генерирането на насочен граф е по-трудна задача за балансиране от генериране на ненасочен граф, защото трябва да генерираме симетрична матрица спрямо обратния диагонал.

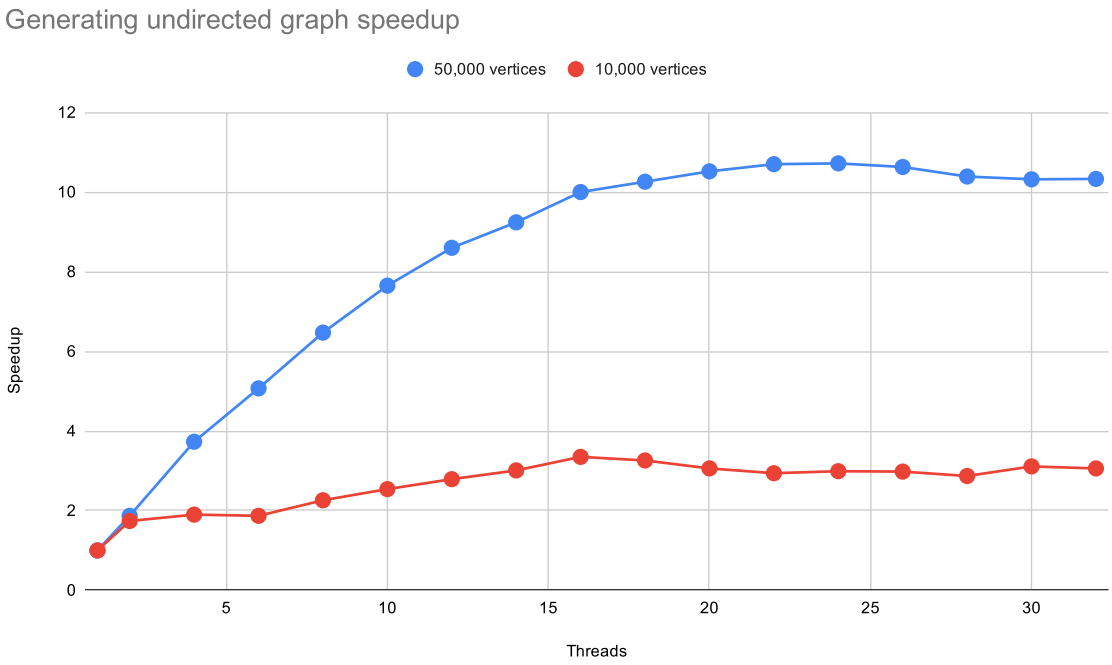
За тази цел може да разделим работата която всяка една нишка върши на попълване на 1 ред и 1 колона, така че матрицата да е симетрична. Това довежда до проблем, че трябва да достъпваме данни извън кеша на процесора (други редове от матрицата), което ще доведе до забавяне. Тъй като това забавяне е непридвидимо не може да разделим работата по равно на всички нишки със статично балансиране. За да се справим с този проблем ще използваме **динамично балансиране**, за да може когато една нишка свърши задачата си максимално бързо да вземе следваща задача, без да чака останалите нишки. За динамичното балансиране отново използваме **Master-slaves** архитектура. Това няма да причини забавяне защото всеки един worker има да извърши много продължителна работа, докато задаването на задачи от Master нишката е много бърза операция.

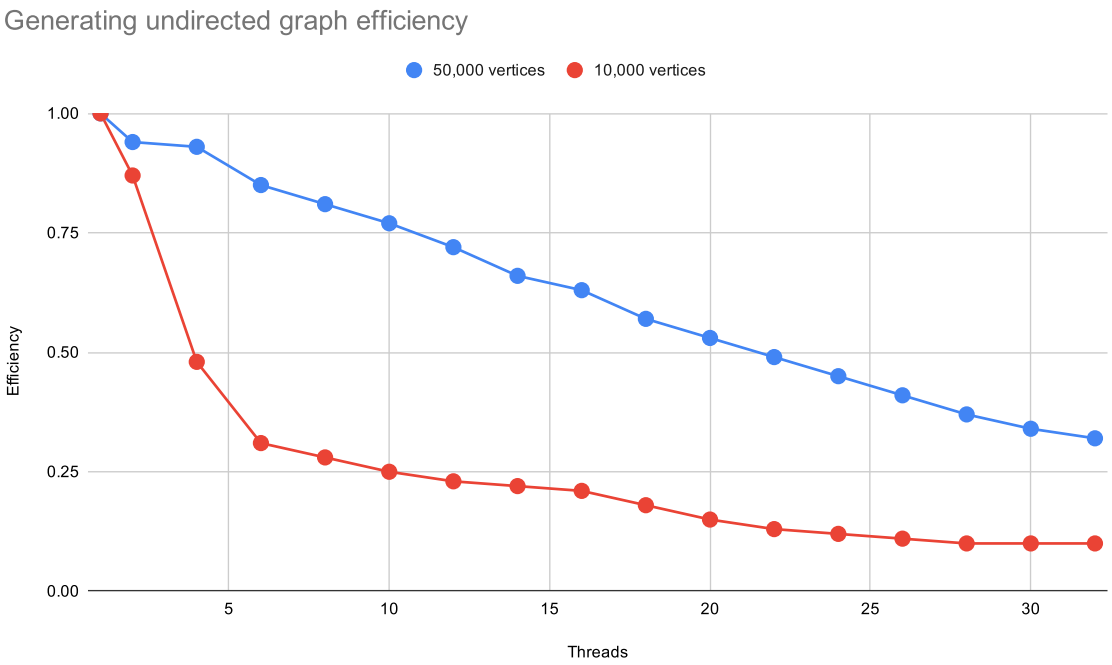
Генериране на ненасочен граф с 10,000 върха

Threads	T1	T2	T3	Tp = min()	Speedup	Efficiency
1	2704	1815	1775	1775	1	1
2	2588	3018	1018	1018	1.74	0.87
4	933	1300	1518	933	1.9	0.48
6	947.85	1202	1077	947.85	1.87	0.31
8	791.27	786.66	809.33	786.66	2.26	0.28
10	697.49	729.61	761.23	697.49	2.54	0.25
12	673.36	662.63	635.66	635.66	2.79	0.23
14	589.05	609.3	632.46	589.05	3.01	0.22
16	529.87	601.13	613.44	529.87	3.35	0.21
18	585.82	543.97	606.32	543.97	3.26	0.18
20	579.42	617.38	595.43	579.42	3.06	0.15
22	608.97	603.57	603.83	603.57	2.94	0.13
24	600.97	592.97	604.98	592.97	2.99	0.12
26	608.3	622.92	594.92	594.92	2.98	0.11
28	630.03	735.92	618.88	618.88	2.87	0.1
30	570.18	582.91	585.46	570.18	3.11	0.1
32	579.31	588.33	592.78	579.31	3.06	0.1

Генериране на ненасочен граф с 50,000 върха (времето е измерено в секунди)

Threads	T1	T2	T3	Tp = min()	Speedup	Efficiency
1	137.01	140.9	144.32	137.01	1	1
2	77.04	73.22	73.54	73.22	1.87	0.94
4	36.71	38.99	39.44	36.71	3.73	0.93
6	27.05	27.22	27.07	27.05	5.07	0.85
8	21.59	21.41	21.18	21.18	6.47	0.81
10	18.24	17.92	18.03	17.92	7.65	0.77
12	16.25	15.94	16.01	15.94	8.6	0.72
14	14.82	15.06	14.82	14.82	9.24	0.66
16	13.7	14.39	14.23	13.7	10	0.63
18	13.51	13.8	13.35	13.35	10.26	0.57
20	13.2	13.02	13.24	13.02	10.52	0.53
22	13.12	13.07	12.8	12.8	10.7	0.49
24	12.78	12.87	12.99	12.78	10.72	0.45
26	13.15	13.31	12.89	12.89	10.63	0.41
28	13.19	13.47	13.41	13.19	10.39	0.37
30	13.75	13.27	13.84	13.27	10.32	0.34
32	13.46	13.26	13.58	13.26	10.33	0.32





Извод

Генерирането на ненасочен граф отново показва добри резултати само при големи графи. Ускорениението е почти 2 пъти по-лошо от колкото при насочените графи. Максималното ускорение, което постигаме е около 11, което за 16 ядрен процесор е приемлив резултат. Причината алгоритъмът да не се справя толкова добре, колкото алгоритъма за насочен граф, е че не се използва кеша на процесора достатъчно ефективно. Много време се губи когато се попълва колоната в матрицата на съседство. Редът в матрицата на съседство е масив и се обхожда много бързо (какво видяхме в алгоритъма за насочен граф).

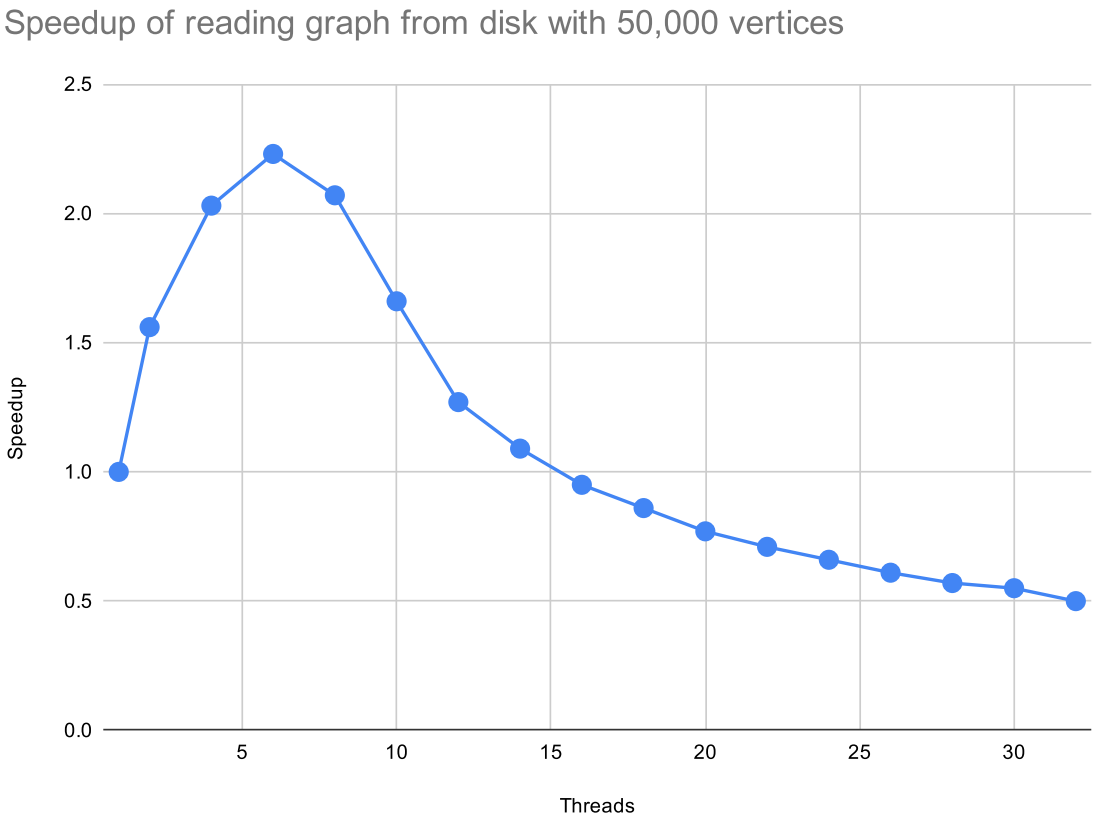
Операция: Прочитане на граф от файл

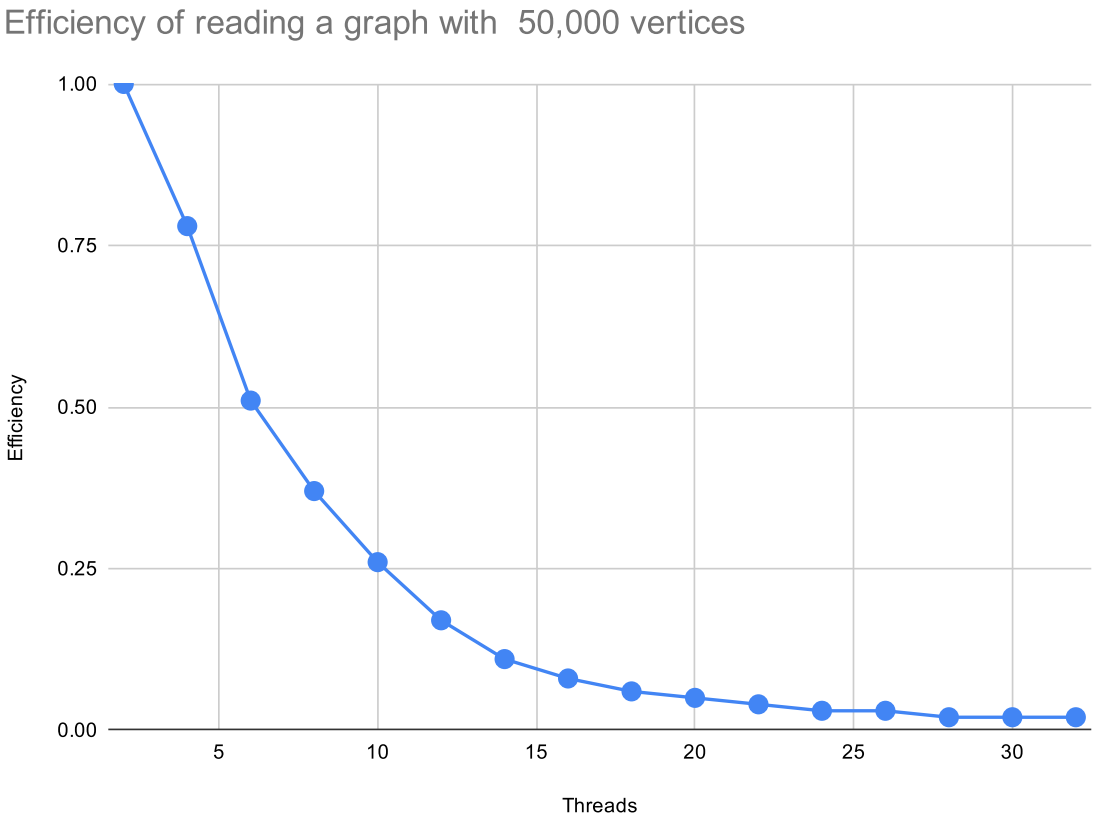
Прочитането на граф от файл е бавна операция и зависи в голяма степен от хард диска с който разполагаме и под каква форма се съхраняват данните. В тестовете описани в този документ данните се съхраняват на един физически диск, не използваме разпределено съхранение на данните. Това довежда до ограничение на ефективността на паралелни алгоритми за прочитане на граф от файл.

За прочитане на граф от файл използваме **статично балансиране** - разпределяме редовете във файла на броя на нишките с които работим. Статично балансиране е възможно защото знаем точният формат на файла, в който ще се съхранява графа. Всяка нишка обработва еднакъв брой редове от файла на принципа на **Round-Robin**. Пример: ако имаме 16 нишки, то нишка 1 обработва редовете от файла които дават модул 1 при деление на 16 (редове 1, 17, 33, ...). По този начин всяка нишка прочита ребрата на равен брой върхове. По този начин може да разпределим прочитането на ред от файла и обработването му (конвертиране до ребра на графа). Забавянето се получава от четенето на данни от диска. Обработването на данните се случва паралелно на всички нишки четат от един и същ файл и трябва да се изчакват.

Прочитане на граф с 50,000 върха (4.7 GB)

Threads	T1	T2	T3	Tp = min()	Speedup	Efficiency
1	14.78	14.22	15.05	14.22	1	1
2	9.13	9.4	9.49	9.13	1.56	0.78
4	7.02	7.03	7.05	7.02	2.03	0.51
6	6.42	6.64	6.37	6.37	2.23	0.37
8	7.06	7.53	6.87	6.87	2.07	0.26
10	8.71	8.57	8.99	8.57	1.66	0.17
12	11.19	11.68	11.3	11.19	1.27	0.11
14	13.15	13.06	12.99	12.99	1.09	0.08
16	14.92	14.95	15.39	14.92	0.95	0.06
18	16.7	16.53	16.66	16.53	0.86	0.05
20	18.73	18.37	18.66	18.37	0.77	0.04
22	19.95	20.1	20.32	19.95	0.71	0.03
24	22.09	21.81	21.53	21.53	0.66	0.03
26	23.26	23.44	23.14	23.14	0.61	0.02
28	24.86	25.15	24.96	24.86	0.57	0.02
30	26.14	27.44	25.8	25.8	0.55	0.02
32	28.24	28.58	28.93	28.24	0.5	0.02





Извод

Четенето от файл, не е операция която може да паралелизираме ефиктивно, когато нямаме подходящо хранилище на данни, а имаме само 1 хард диск. Оптимално време за четене от файл постигаме при 4 или 5 нишки - малко повече от 2 пъти по-бързо от четене с една нишка, но при 5 нишки ефективността е едва 50%.

Потенциално подобрение

Ако използваме разпределено съхранение на данните като например sharding е възможно да постигнем по-добри резултати.

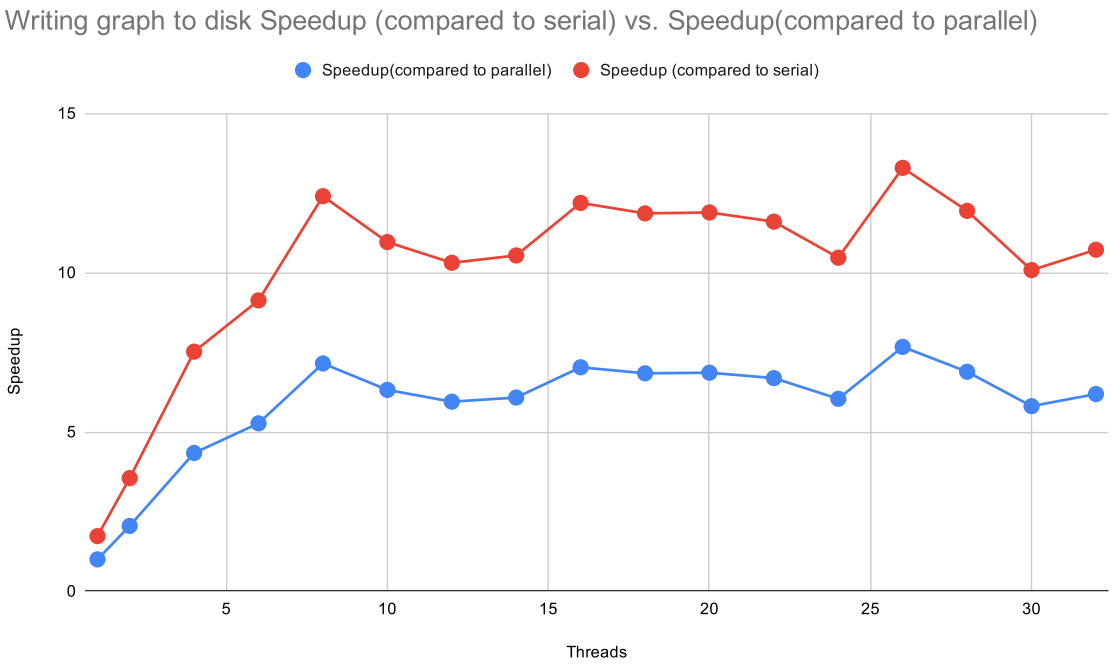
Операция: Записване на граф във файл

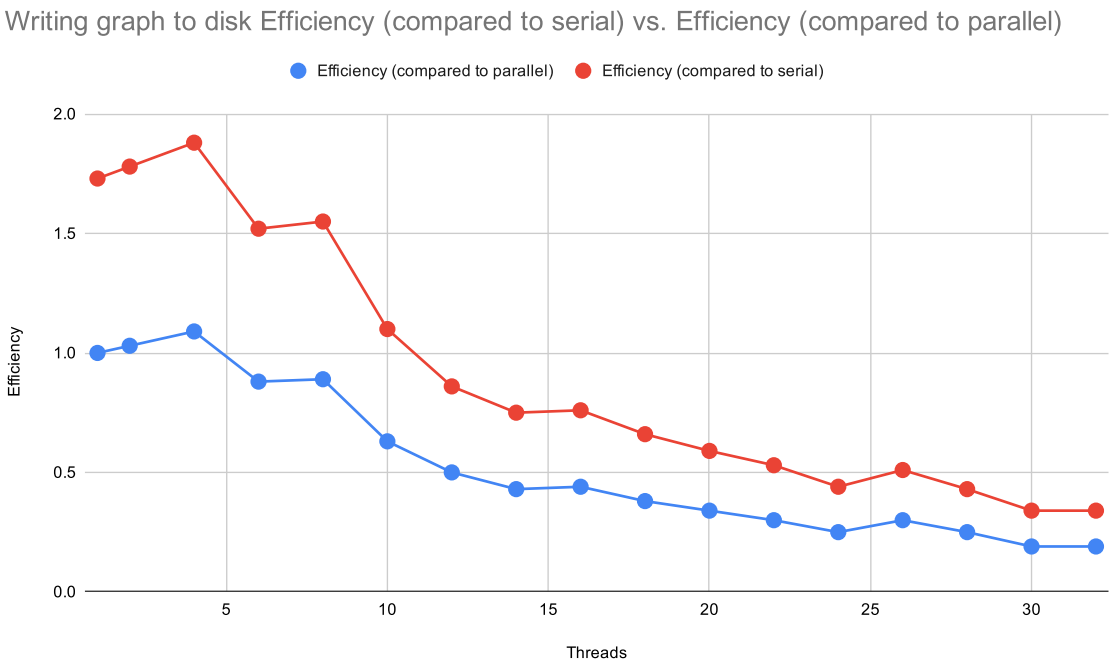
Записване на граф във файл отново зависи от хардуера и важат същите разсъжцения както за Прочитане на граф от файл.

За разлика от прочитането на граф от файл, тук ще използваме **динамично балансиране**, за да направим сравнение на двата подхода. Идеята на алгоритъма е да стартира **Worker Pool**. След което започва да дава задания на този pool. Всяко задание се състои в конвертирането на съседите на даден връх до масив от байтове, който е подходящ за записване във файл. След като някой worker изпълни заданието си изпраща съобщение до основната нишка със съобщение че съседите на връх X са готови да бъдат записани във файла. Основната нишка проверява дали всички върхове до X-1 са записани и ако да - записва съседите на X, ако не - добавя X във опашка с готови върхове чакащи да бъдат записани. Използването на **Master-slaves** подход за алгоритъма не пречинава забавяне при малък брой ядра на процесора и голям брой върхове, защото добавянето на ново задание е много бързо, а worker-нишките имат много информация за обработване. т.е забавяне има само в началото, докато всяка нишка получи по 1 задание, но това забавяне е от порядъка на (няколко наносекунди) * (броя нишки).

Записване на граф с 50,000 върха (4.7 GB)

Threads	T1	T2	T3	Tp = min()	Speedup(compared to parallel)	Efficiency (compared to parallel)	Speedup (compared to serial)	Efficiency (compared to serial)
1	68.88	65.11	63.79	63.79	1	1	1.73	1.73
2	34.55	36.9	31.12	31.12	2.05	1.03	3.55	1.78
4	14.7	18.72	15.75	14.7	4.34	1.09	7.52	1.88
6	17.07	12.11	12.71	12.11	5.27	0.88	9.13	1.52
8	10.89	10.86	8.92	8.92	7.15	0.89	12.4	1.55
10	15.28	14.98	10.09	10.09	6.32	0.63	10.96	1.1
12	12.69	16.82	10.72	10.72	5.95	0.5	10.31	0.86
14	18.24	10.51	10.49	10.49	6.08	0.43	10.54	0.75
16	15.32	12.59	9.07	9.07	7.03	0.44	12.19	0.76
18	11.35	9.32	9.99	9.32	6.84	0.38	11.86	0.66
20	10.53	10.28	9.3	9.3	6.86	0.34	11.89	0.59
22	9.53	12.59	12.79	9.53	6.69	0.3	11.6	0.53
24	10.61	12.25	10.56	10.56	6.04	0.25	10.47	0.44
26	9.67	11.65	8.32	8.32	7.67	0.3	13.29	0.51
28	9.26	9.9	10.83	9.26	6.89	0.25	11.94	0.43
30	10.97	11.21	12.12	10.97	5.81	0.19	10.08	0.34
32	10.31	11.43	11	10.31	6.19	0.19	10.72	0.34





Извод

За разлика от четенето на гракф от диск, при записването на граф на диска е възможно да постигнем до 10/12 пъти по-добро време използвайки многонишкова обработка. Сравненията с паралелният алгоритъм ни дават по-лошо забързване защото в основата си той работи поне с 2 нишки и няма лесен начин да го лимитираме. Например с 8 нишки постигаме доста добро ускорение и ефективност. Значително подобрение над непаралелния алгоритъм.

Потенциално подобрение

Ако използваме разпределено съхранение на данните като например sharding е възможно да постигнем по-добри резултати.

Използване на програмата

За стартиране на програмата и извършване на допълнителни тестове е необходимо да има инсталиран компилатор на Go на машината на която се тества.

Възможни аргументи

- -v N - генериране на граф с N върха
- -t N - използване на N нишки. При подаване на 0 нишки програмата сама преценява колко нишки да използва спрямо това колко процесора има машината
- -d N - плътност на графа в проценти $N \in [0, 100]$
- -q - изпълнение на програмата в тих режим (извежда по-малко съобщения за изпълененението си)
- -directed - генериране на насочен граф (по подразибиране е ненасочен)
- -g - само генериране на граф и запаметяване във файл, без обхождане
- -i myFile.graph - прочитане на граф от файл myFile.graph и обхождане
- -o myFile - специфициране на името на файла в който да се запише изхода от програмата

Примерно използване

Генериране на насочен граф със 100 върха, 30% density, и запаметяването му във файл с име myGraph.graph. Тъй като не са зададени брой нишки, програмата сама ще определи броя нишки в зависимост от това с колко нишки разполага процесора/ите на машината.

```
go run bfs.go -g -v 100 -o 'myGraph' -d 30 -q -directed
```

Прочитане на граф от файл myGraphFile.graph и изпълняване на обхожданията със 64 нишки.

```
go run bfs.go -i 'myGraphFile.graph' -t 64
```

Бъдещи планове

В този документ са разгледани алгоритми за траверсиране на граф в ширина.. Би било интересно да се изследва дали при такива графи обхождане в дълбочина може да даде по-добри резултати. Може да се разгледа и различно представяне на графа и дали това оказва влияние - вмест матрица на съседство да се използва списък на съседство за представяне на графа. Дали различното представяне ще даде отражение върху ускорението, което получаваме при графи с по-малко върхове. Има и други алгоритми за разпределяне на работата при обхождане в ширина, а именно - 2-D partitioning, които не са тествани в този документ.

Източници

[1] A scalable distributed parallel breadth-first search algorithm on BlueGene/L., Yoo Andy, et al. Proceedings of the 2005 ACM/IEEE conference on Supercomputing. IEEE Computer Society, 2005.

[2] Level-Synchronous Parallel Breadth-First Search Algorithms For Multicore and Multiprocessor Systems, Rudolf, and Mathias Makulla FC 14 (2014)

[3] "Parallel breadth-first search on distributed memory systems.", Buluç, Aydin, and Kamesh Madduri. Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011.

[4] A Tale of BFS: Going Parallel, Egon Elbre

[5] Will Hyper-Threading Improve Processing Performance?, Bill Jones, Sr. Solution Architect, Dasher Technologies