Курсов проект за дисциплина

“Системи за паралелна обработка”

ODDLY-Even Sort

Методи за паралелизиране на odd-even sort и техническите им характеристики.

Изготвил: Румен Азманов, ФН:82176, КН, Курс 3

Под ръководството на: Проф. д-р Васил Цунижев

Съдържание

Увод

В програмирането алгоритмите за сортиране са основен клас решения на задача за наредба на елементи в определена структура. Всеки алгоритъм освен и различен метод на обработка на данните притежава и специфични характеристики. Най-често за разграничение се използва характеристиката за асимптотична сложност на алгоритъма, понеже се цели практическа приложимост в реални условия. Това е и характеристиката, която се разглежда и при паралелизиране на една програма. В случая се изследва обаче не само скоростта, но и другите свойства на алгоритъма - дали е стабилна сортировка, за какви данни се прилага, дали е оптимизирана за системи с по-голям кеш и т.н.

Цел на проекта

Целта на проекта е да се разгледа алгоритъмът “odd-even sort” (brick sort) и не само методите за неговото ускорение, а и начините, по които паралелизацията влияе на неговите свойства. Ще се реализират различни негови вариации и ще се срвани ефективността по време.

Odd-even sort се характеризира с квадратична сложност $O(N^2)$ по време и константна O(1) сложност по памет. Спада в групата на сортировките, които са приложими за всякакъв вид данни, понеже използва метода на сравнение и размяна на елементи. В стандартната си реализация е стабилен (не разменя еднакви елементи спрямо първоначалната наредба), може да бъде реализиран адаптивно (да работи по-бързо за определени наредби) и ако е приложен над подходяща структура от данни ще се приложи и оптимизацията на cache locality. Особено важно е да се отбележи, че е лесно да се имплементира паралелно.

Odd-even sort се състои от последователно итериране на данните и сравняването на тези на четен индекс със следващ елемент, а след това аналогично за тези на нечетен индекс. Операциите се редуват до N пъти, където N е брой елементи в масива. Поради тази причина алгоритъмът е с квадратична сложност.

Анализ

В процеса на реализация на проекта бе необходимо да се анализират множество източници на информация. В някои от тях се разглеждат и по-различни алгоритми за сортиране - bubble sort, odd–even mergesort и други. От тези източници главно се разглеждат идеите за метод за паралелизация на задачата.

Анализ на методите за паралелизация

В източник [1] се коментира реализацията на алгоритъма bubble sort чрез паралелизиране на масив и разделяне на части спрямо броя нишки. При него целят масив се разделя на части, броят на които е равен и на броя нишки, които могат да се ползват. Всяка нишка има за основната си задача да итерира през данните, прилагайки сравняване и размяна на съседни елементи. Същият метод се прилага и в решението в този проект. Odd-even sort също безпроблемно се паралелизира, чрез разделяне на даден масив, понеже в първата си стъпка се сравняват само съседни елементи, тоест няма пряка зависимост между данни, които се намират на твърде отдалечени места в масив. Понеже на практика при итерация над нечетни и четни елементи вече съществува пряка зависимост, то всяка нишка може да итерира и по двата вида елементи последователно. Тук се посочва и синхронизационният проблем в източник [1]. Всяка нишка има пресичащи се елементи със следващата и с предишната (където “следваща” и “предишна” са само термини по декомпозирането на данните). Основаното решение е при тези елементи да се осъществи синхронизация между нишките.

В източник [4] е представен алгоритъм за сортиране с използване на графичен процесор. Псевдокодът разделя паралелизацията на нишки, които работят на една и съща стъпка от сортирането на масива. В проекта е представено и такова решение, при което на всяка стъпка се изпълняват всички нишки и те работят паралелно, но за изпълнението на следващата стъпка (слой на мрежата) се изисква те да се синхронизират. Това решение е твърде бавно и не спазва добрите практики на писането на конкурентен код, понеже изисква твърде много етапи на паралелизация и не е оптимизирано за кеша на процесорите. Поместено е в проекта само като етап на разработка, за да се ползва за анализ и промяна.

Анализ на методите за синхронизация

При синхронизацията между нишките в източник [1] се ползват споделени условни променливи между нишките. За целта е необходимо всяка от тях да запазва на коя стъпка от алгоритъма се намира и да изчаква останалите да са на същата стъпка. В проекта се използва също аналогичен метод, но за целите на езика и реализацията изчакването е не чрез системни примитиви за сигнализация или mutex структури, а само чрез системно извикване на текущата нишка за реорганизация на ресурсите. Синхронизацията в източник [4] се очаква да се извърши с приключването на работата на нишките върху всяко от заданията.

Анализ на зависимостите

От дефиницията на разглеждания алгоритъм следва, че между всеки две итерации (четна и нечетна) има пряка зависимост между данните, но при самата итерация няма зависимости. В източник [2] се разглеждат сортиращи мрежи и начините на разделяне на операции на етапи, като в самите етапи няма пряка зависимост между данните или ако има такава, то се изпълняват последователно дадените операции от една и съща нишка. Реализацията на алгоритъма съставя етапи от по 2 последователни итерации (само разпределената част) на всяка нишка и синхронизация да не се надвишава поредността на итерация на нишка спрямо двете съседни по данни на нея с повече от едно ниво. При алгоритъма с разделяне на всеки етап на нишки и синхронизирането след края няма такава възможност за изпълняване на повече от една стъпка от една нишка, понеже със сигурност зависи от останалите елементи, които не се обработват от нея.

Анализ на методите за декомпозиция на данните/функциите и грануларността

При декомпозицията на масива на отделни последователности за всяка нишка се постига оптимално статично разпределение на проблема за извършване на сравнения. Понеже за задачата е неизвестно статистическото разпределение на данните, то не би било ефективно да се търси начин за динамично балансиране. Освен това се цели ползване на малки по размер прости типови стойности, които да се възползват от нивата на кеш на процесорите. Функционалното разпределение също не се среща като оптимално решение в повечето източници, понеже задачите за сортиране често се състоят от изпълнението на сходни операции над различни данни наместо разпределяне на ресурсите. Така и всички алгоритми в проекта се базират на основна функция, която да разпредели ресурсите и да контролира създаването и изпълнението на нишките (в някои случаи се явява синхронизираща, но не във всички).

Поради причината, че декомпозирането се базира пряко на броя нишки, които трябва да реализират алгоритъма, то не разглеждаме пряко стойности за грануларност. Те се получават от общия размер (брой данни), разделени поравно на всяка нишка.

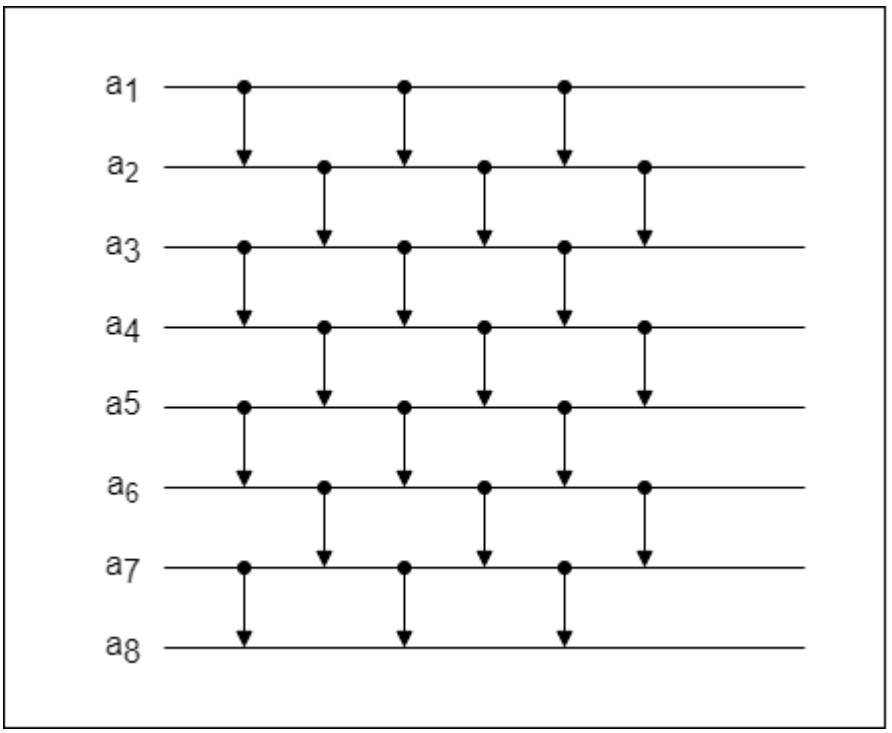
Анализ на адаптивността на алгоритъма и използването на кеша

За текущия проект е особено важно да се разгледат и абстрактните и техническите свойства на алгоритмите.

Стандартният odd-even sort изпълнява поне N пълни (четни и нечетни) итерации над масив от N стойности. Подобно решение често се явява излишно усложнено. В източник [5] изрично е доказана и коректността на адаптивна вариация на алгоритъма. Подобна реализация е и един от най-често срещаните варианти на алгоритъма. При нея на всеки етап се проверява дали въобще е изпълнена размяна след сравнение и

ако не е, то алгоритъмът прекратява своята работа. Подобно свойство на алгоритмите ги прави адаптивни и означава, че те могат да имат линейна сложност за сортирани последователности. В този проект не се търси идеален начин да се приложи това свойство и за паралелни решения, а главно да се сравни за какви данни има някакво значение.

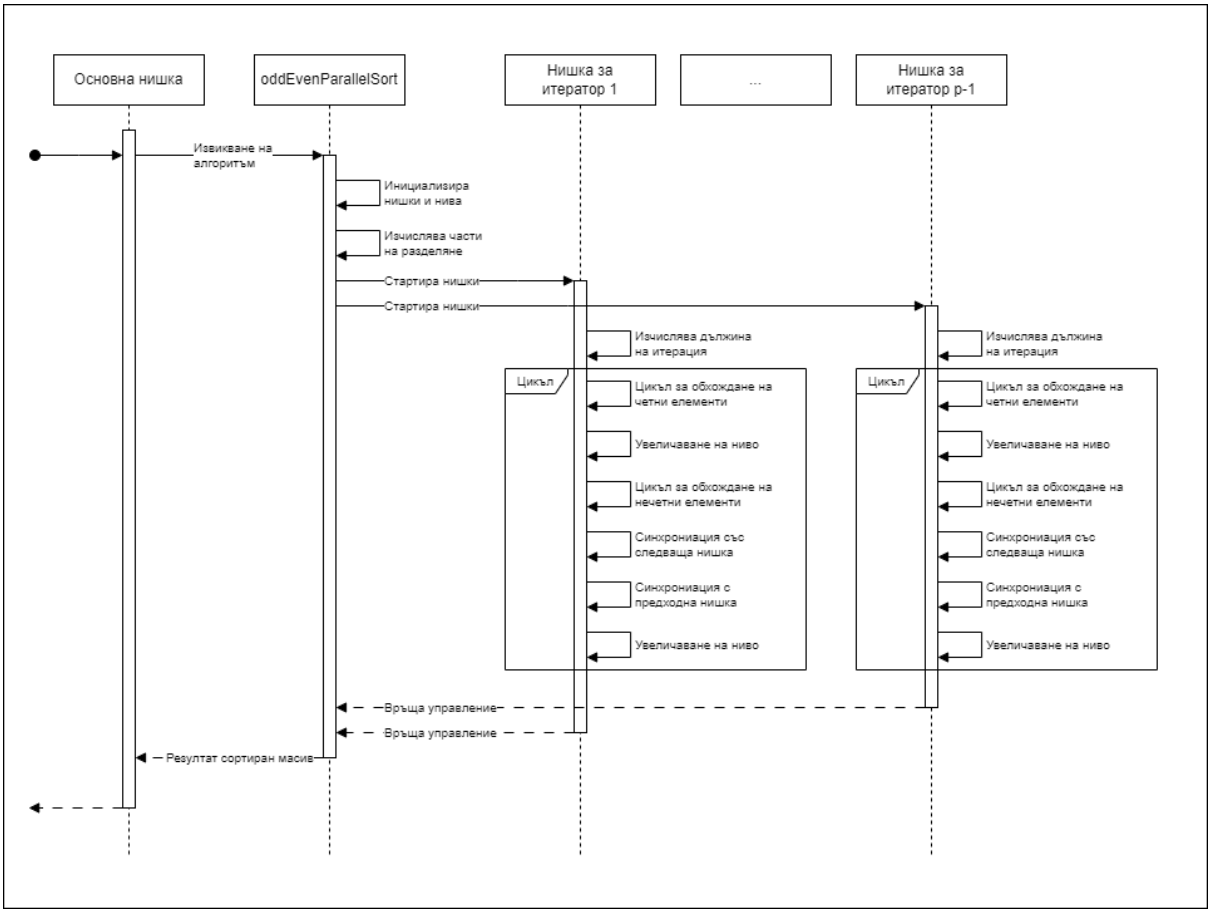
Специфично техническа характеристика е използването на кеша на процесора. За да се разгледа влиянието на това свойство във всички тестове се използва проста последователна структура от данни - масив. При основния алгоритъм за разделяне на парчета се очаква да се наблюдава различие в бързодействието при ползване на различни типове данни за едни и същи спецификации на кеша на процесора. Не за всеки реализиран алгоритъм обаче това ще е от влияние, понеже всеки от тях обхожда по различен начин данните.



Сортираща мрежа за odd-even sort

Модел на програмата

В текущия проект основното решение е разпределянето на задачата по сортиране на масива чрез разпределянето на части от масива на всяка нишка. Диаграмата описва извикването на сортиращата функция, която последователно задава на всяка нишка отделна част и на всеки етап на зависимост те сами изпълняват синхронизация помежду си чрез изчакване.



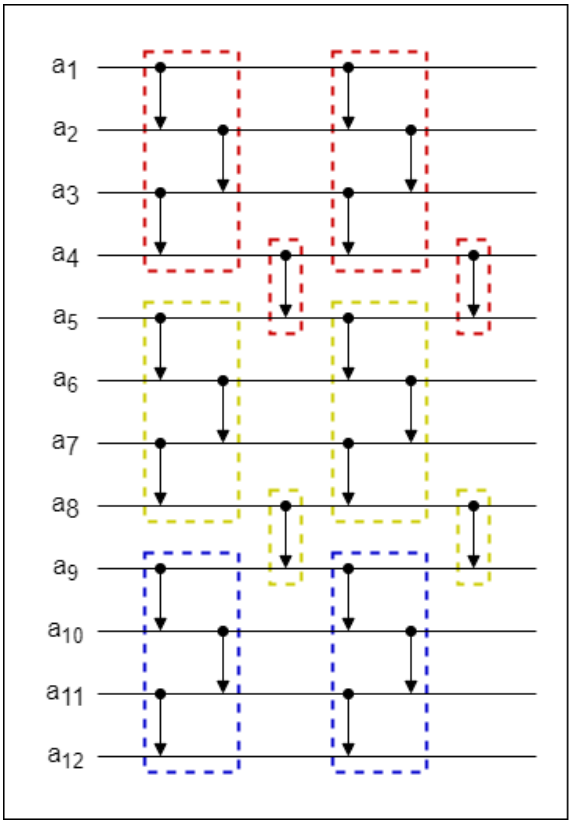
Цялостната програма се състои освен от този основен алгоритъм и от други, с които да се сравнява паралелното изпълнение. Основен е горепосоченият (oddEvenParallelSort), който реализира стандартната имплементация на odd-even sort за една нишка. Добавени са и други варианти:

- Адаптивна версия на алгоритъма, която да се ползва за сравняване на зависимости със стойностите в данните. (oddEvenSort)
- Имплементация на сортирането със стъпка на обединяване на резултатните масиви. Рекурсивен, прилага се паралелизъм. (oddEvenMergeSort)
- Паралелна версия с разделяне на всяка итерация на отделни нишки, при което те трябва да се синхронизират. Не се ползв при тестовете, понеже е твърде бавна, служи само за анализ по време на разработката. (oddEvenParallelSortJump)

- Сортиране, при което са използвани SIMD функции за по-бърза обработка. Имплементацията е само за int. (oddEvenSIMDSort)

Има и единствена обща тестова функция, която генерира данни и засича време за всяка от функциите. При нея се изпълняват по три теста за всяка сортировка и ако е необходимо се ползва и определен брой нишки в тестването. Изолзват се и някои други малки помощни функции за работата на програмата. Една от тях реално реализира темплейтен comparator от сортираща мрежа.

На графиката е демонстриран модел на работа на основния алгоритъм с 3 нишки. Всеки цвят е отделна нишка, която изпълнява определените операции. След приключване на основния блок сравнения е необходимо всяка нишка да се синхронизира със следващата и предишната. Това става като единичното сравнение може да настъпи само ако съседните по разпределение нишки са завършили с основната част на своите сравнения.



Примерно действие на основния алгоритъм чрез сортираща мрежа за 3 нишки

Технологично проектиране

Технологични примитиви

Основният алгоритъм за сортиране *oddEvenParallelSort* се реализира чрез разпределяне на динамичен масив (*vector*) от данните и неговият размер (подава се и размер на първо място за гъвкавост на употребата и заради необходимостта от тестването над едни и същи случайно генерирани данни). Всяка нишка се реализира чрез статичен масив от нишки, за всеки елемент на който се извиква *std::thread* с необходимите параметри:

```
std::thread threads[64];
for (int i = 0; i < numberOfThreads; ++i) {
  if (i != numberOfThreads - 1) {
    threads[i] = std::thread(oddEvenParallelIteration<T>,
      std::ref(dataVector), size, (i * split), split, phases, i,
      numberOfThreads);
  } else {
    threads[i] = std::thread(oddEvenParallelIteration<T>,
      std::ref(dataVector), size, (i * split), (size - i * split), phases,
      i, numberOfThreads);
  }
}
```

Подават се и общо споделени масив от фази и общ брой нишки за нуждите на изчакването на нишките помежду си. Всяка от тях проверява предходната и следващата по наредба на частите в масива и изчаква до съвпадение на етапите с тях. През това време се изпълнява *std::this_thread::yield()* за системно изискване приоритетът на нишките да се преподреди. По този начин, освен че всяка нишка на практика изпълнява spinlock техника и своевременно приоретизира работата на останалите нишки. Отбелязва се, че не се използват бавни за изпълнение примитиви като mutex или променливи с условие, които да забавят бързодействието на прогарамата.

```
if (threadID < numberOfThreads - 1) {
  while (phases[threadID + 1] < phases[threadID]) {
    std::this_thread::yield();
  }
  comparator(dataVector[end], dataVector[end + 1]);
}
if (threadID > 0) {
  while (phases[threadID - 1] <= phases[threadID]) {
    std::this_thread::yield();
  }
}
```

Позитивното в този алгоритъм е, че ако масивът се раздели на определен брой нишки (съответно и на толкова части), то синхронизацията няма да се извършва на всяка отделна част, както би се очаквало, а ще се използва свойството на odd-even sort стъпките, при която понеже всяка зависи непосредствено от предишната, то може две стъпки (четна и нечетна) за една и съща част да се изпълнят от една и съща нишка. Тоест за масив от N елемента, разпределен на P нишки, а именно N/P части, които се изпълняват за N/2 стъпки (свойство на алгоритъма), ще имам само N/4 места за синхронизация на тези P нишки.

Алгоритъмът за сортиране по метод на SIMD операции ползва съответно необходимите SSE команди за работа с тип int. Останалите типове не са реализирани, понеже не всяка система (компилатор) позволява ползването на такива операции и се изискват твърде много допълнителни настройки.

Технологични примитиви за тестване

При изпълнението на програмата са зададени параметрите за тестване с нишки и размери на данните. За тестването е необходимо само да се изпълни компилиран изпълним файл. За вътрешно генерираните случайни данни се използва стандартна за езика C++ функция *rand()*. Времето на работа на всеки един тест също се засича с примитиви на езика, които са *std::chrono::steady_clock::time_point* и отчитат два момента (непосредствено преди и след извикване на сортировка) и разликата между тях. Задължително след всяка сортировка се извършва и тест за коректност на данните. Коректността между самите тестове се гарантира чрез употреба на едно и също множество стойности. Единствена разлика е ползването на int, char и double като типове за данните за възможност да се сравнят данни с различен размер.

Процедура по тестване

За тестването на времената на алгоритмите са ползвани 2 тестови машини. В зависимост от наличните ресурси на системите и поддържаните от съответен компилатор ресурси системите са:

Система	PC	Server
Процесор	Intel(R) Core(TM) i5-6300U	Intel(R) Xeon(R) CPU E5-2660
	2.50GHz	2.20GHz
L1 cache	128K	32K + 32K
L2 cache	512K	256K
L3 cache	3000K	20480K
Сокети	1	2
Ядра	2	32

При тестването се генерират случайни данни чрез системни примитиви. Те са от тип int, char или double. Няма определени статистически разпределения на данните и се ползва една и съща наредба за всички тестове за да бъдат стандартизирани. В дадените тестове са ползвани средите за разработка и компилаторите MS Visual Studio и GCC 8.1.0 на операционни системи съответно Windows и Linux. С изключение на поддръжката на SSE функционалности няма разлика в тяхната компилация. Използван е -O3 флаг за максимално ниво на оптимизация в Release режим.

Тестови резултати

В този документ е поместена само една от таблиците за тестване на проекта и алгоритмите. Във всяка таблица обаче са означени: номер на тест, n - размер на данните, алгоритъм, който е използван, p - брой нишки за тестване, $T_{\square}^{(i)}$ - пореден тест за проверка с p нишки, минимална стойност на тест, $S_{\square}$ - постигнато ускорение чрез отношението на изпълнението на една нишка и p нишки, $E_{\square}$ - ефективност на паралелното изпълнение.

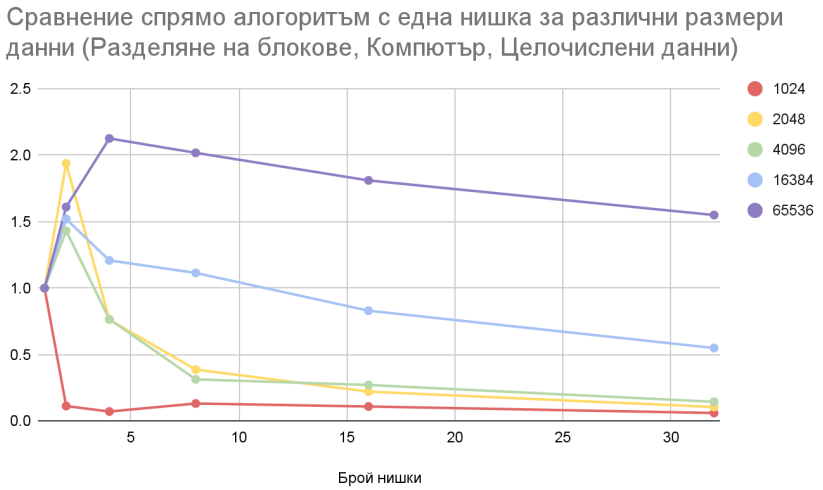
Тест 1, Система Домашен компютър, За данни от тип INT

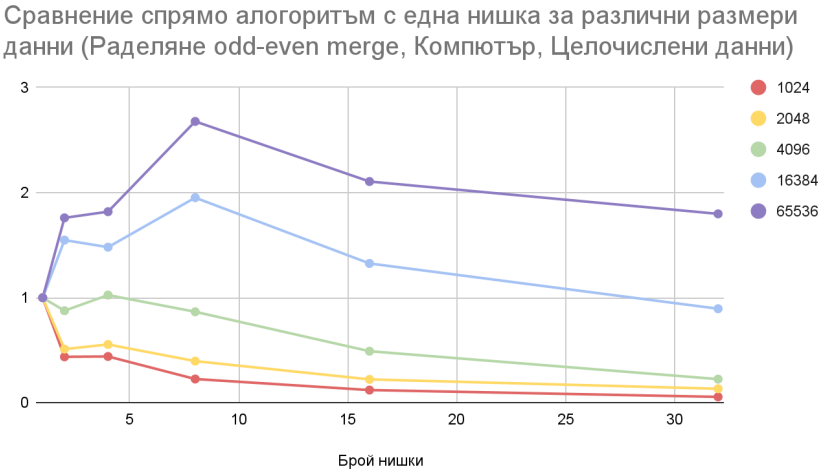
#	n	Алгоритъм	p	$T_{\square}^{(1)}$	$T_{\square}^{(2)}$	$T_{\square}^{(3)}$	$T_{\square}=\min(T_{\square}^{(i)})$	$S_{\square} = T_1/T_{\square}$	$E_{\square} = S_{\square}/p$
1	1024	Разделяне на блокове	1	2664200	1642700	1386300	1386300	1.00000	1.00000
2			2	12263900	15439600	15388900	12263900	0.11304	0.05652
3			4	28402200	23129900	19298200	19298200	0.07184	0.01796
4			8	10480300	16974000	10530400	10480300	0.13228	0.01653
5			16	13638100	13322400	12653000	12653000	0.10956	0.00685
6			32	22779900	25230900	45559700	22779900	0.06086	0.00190
7		Алгоритъм за odd-even merge sort	1	222800	228700	257600	222800	1.00000	1.00000
8			2	819800	508700	540600	508700	0.43798	0.21899
9			4	514500	678000	504900	504900	0.44128	0.11032
10			8	1100100	982200	1937700	982200	0.22684	0.02835
11			16	2440200	1830200	3106600	1830200	0.12174	0.00761
12			32	4243100	4051600	3915500	3915500	0.05690	0.00178
13		Адаптивен	1	1228100	1015500	1007900	1007900		
14		SIMD операции	1	521200	574200	413600	413600		
15	2048	Разделяне на блокове	1	6280600	6148800	6353900	6148800	1.00000	1.00000
16			2	4003700	3172800	5416800	3172800	1.93797	0.96899
17			4	8062000	8744800	9140000	8062000	0.76269	0.19067
18			8	15872500	25424500	24062200	15872500	0.38739	0.04842
19			16	123089200	31152000	27738300	27738300	0.22167	0.01385
20			32	160403400	58212700	131474100	58212700	0.10563	0.00330
21		Алгоритъм за odd-even merge sort	1	452600	564100	449900	449900	1.00000	1.00000
22			2	1076400	912900	879500	879500	0.51154	0.25577
23			4	929300	810000	5141600	810000	0.55543	0.13886
24			8	2041500	2666500	1132900	1132900	0.39712	0.04964

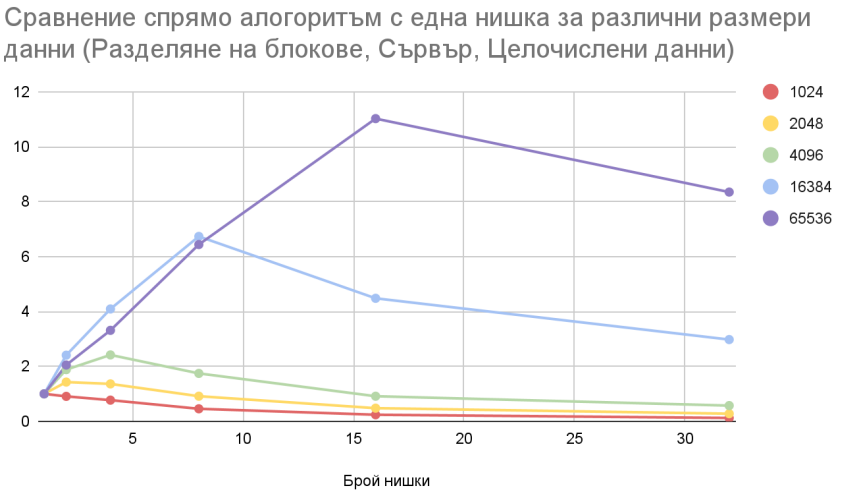
25			16	2016600	3143800	3397900	2016600	0.22310	0.01394
26			32	3387700	4658300	3350000	3350000	0.13430	0.00420
27		Адаптивен	1	4095400	4023900	3926000	3926000		
28		SIMD операции	1	1959600	1837900	1814100	1814100		
29	4096	Разделяне на блокове	1	20901900	15768200	20743400	15768200	1.00000	1.00000
30			2	57383500	51365000	11014300	11014300	1.43161	0.71581
31			4	20571000	25268200	73298300	20571000	0.76653	0.19163
32			8	107266600	50248100	65800700	50248100	0.31381	0.03923
33			16	146075100	231120300	58043200	58043200	0.27166	0.01698
34			32	170086800	108637800	134614700	108637800	0.14514	0.00454
35		Алгоритъм за odd-even merge sort	1	987600	1103100	1261000	987600	1.00000	1.00000
36			2	2034200	1134500	1126900	1126900	0.87639	0.43819
37			4	963200	1883900	1917700	963200	1.02533	0.25633
38			8	2768100	1231700	1139900	1139900	0.86639	0.10830
39			16	2351500	2010100	2260500	2010100	0.49132	0.03071
40			32	5771200	4374700	5207800	4374700	0.22575	0.00705
41		Адаптивен	1	14253200	16200400	13780200	13780200		
42		SIMD операции	1	4119600	7966100	4121500	4119600		

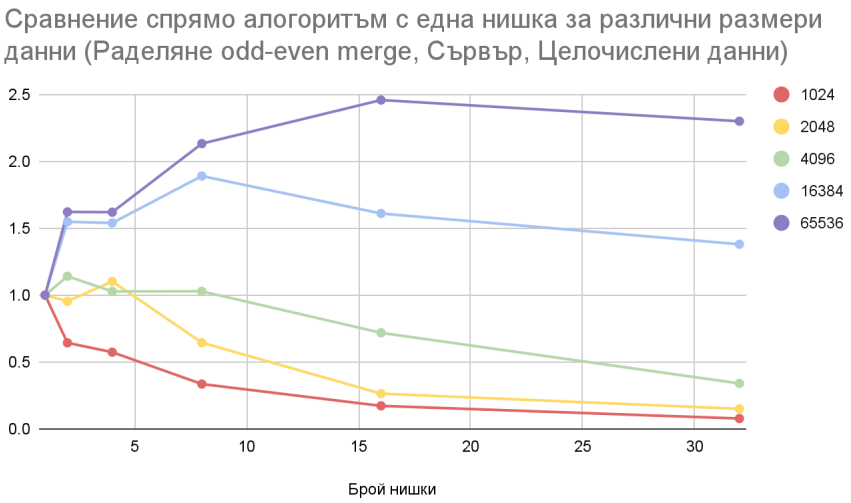
43	16384	Разделяне на блокове	1	332296100	346063900	299061800	299061800	1.00000	1.00000
44			2	627243600	382020800	196779300	196779300	1.51978	0.75989
45			4	398760200	390707200	247447500	247447500	1.20859	0.30215
46			8	346746600	389391000	268348600	268348600	1.11445	0.13931
47			16	477709600	360129400	441099000	360129400	0.83043	0.05190
48			32	544319300	550553400	664565300	544319300	0.54942	0.01717
49		Алгоритъм за odd-even merge sort	1	5472200	5706200	5629100	5472200	1.00000	1.00000
50			2	5240000	3534800	5040200	3534800	1.54809	0.77405
51			4	3775300	4399100	3695400	3695400	1.48081	0.37020
52			8	2804100	4044000	4571400	2804100	1.95150	0.24394
53			16	4127200	4925100	5023500	4127200	1.32589	0.08287
54			32	7680600	6112300	6371700	6112300	0.89528	0.02798
55		Адаптивен	1	342446900	349376600	351062900	342446900		
56		SIMD операции	1	125925000	126395900	128248700	125925000		
57	65536	Разделяне на блокове	1	5610202600	5008376900	5033510300	5008376900	1.00000	1.00000
58			2	3152908100	3158416800	3110072100	3110072100	1.61037	0.80519

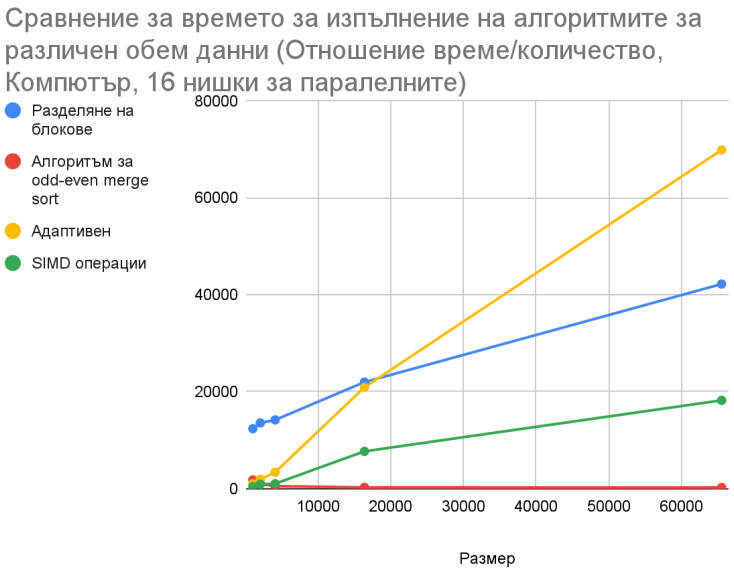
59			4	2377607600	2355876300	2382040000	2355876300	2.12591	0.53148
60			8	2486726500	2481958200	2483463400	2481958200	2.01791	0.25224
61			16	2774922000	2766666300	2796306900	2766666300	1.81026	0.11314
62			32	3231586800	3251526500	5112900200	3231586800	1.54982	0.04843
63		Алгоритъм за odd-even merge sort	1	31294400	34088900	31876300	31294400	1.00000	1.00000
64			2	18055500	18650200	17785200	17785200	1.75958	0.87979
65			4	21668700	18158800	17209900	17209900	1.81840	0.45460
66			8	18564900	11691500	19707100	11691500	2.67668	0.33458
67			16	19709200	21813400	14865700	14865700	2.10514	0.13157
68			32	17407900	17469000	19034900	17407900	1.79771	0.05618
69		Адаптивен	1	5009993700	4581470800	4594106500	4581470800		
70		SIMD операции	1	1220631100	1254338800	1193586500	1193586500		

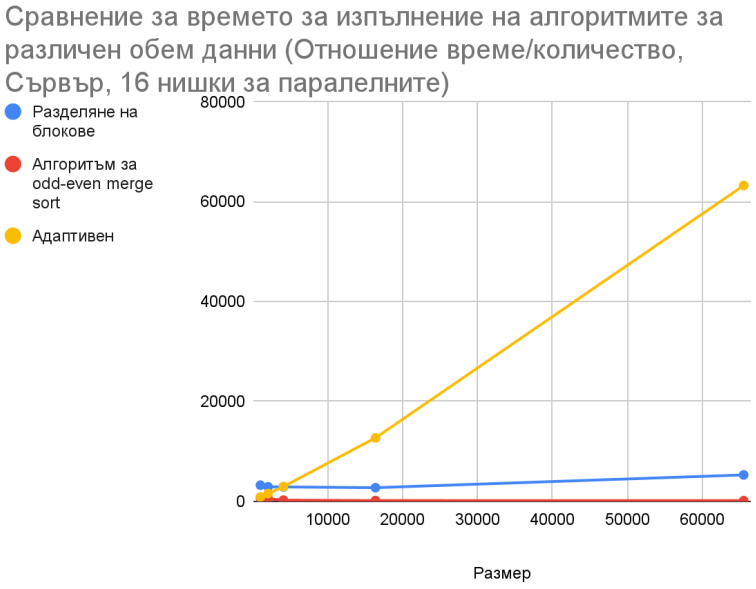


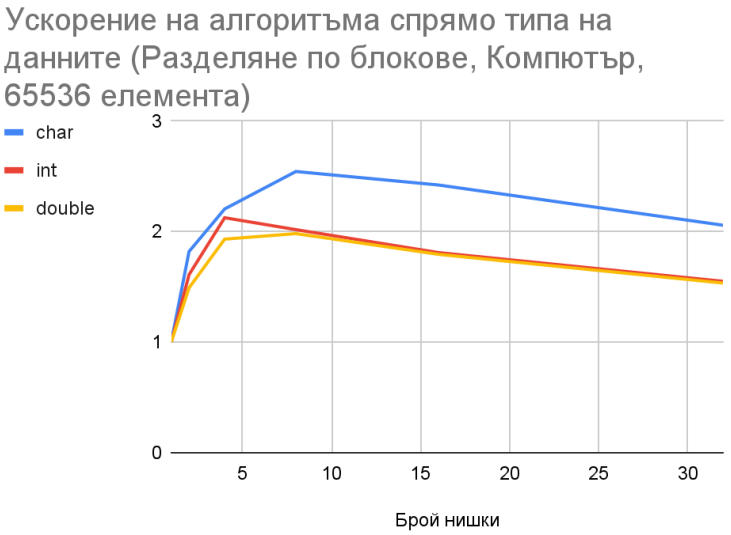


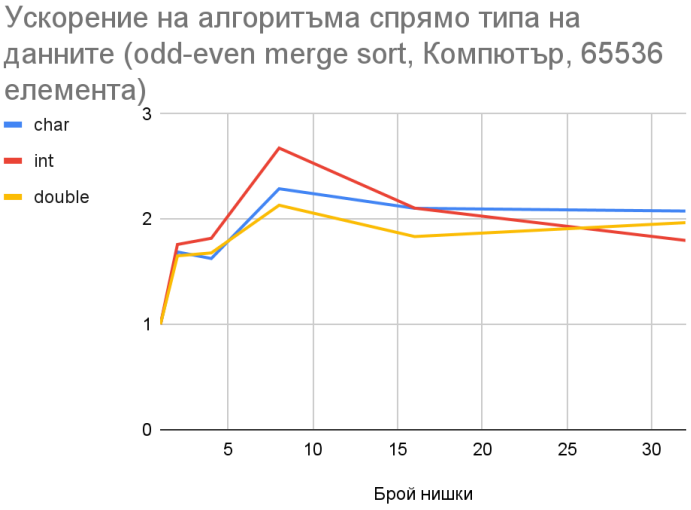

Сравнение спрямо алогоритъм с една нишка за различни размери данни (Разделяне на блокове, Сървър, Целочислени данни)
12
10
8
6
4
2
0
5
10
15
20
25
30
Брой нишки
1024
2048
4096
16384
65536


Сравнение спрямо алогоритъм с една нишка за различни размери данни (Раделяне odd-even merge, Сървър, Целочислени данни)
2.5
2.0
1.5
1.0
0.5
0.0
5
10
15
20
25
30
Брой нишки
1024
2048
4096
16384
65536


Сравнение за времето за изпълнение на алгоритмите за различен обем данни (Отношение време/количество, Компютър, 16 нишки за паралелните)
Разделяне на блокове
Алгоритъм за odd-even merge sort
Адаптивен
SIMD операции
80000
60000
40000
20000
0
10000
20000
30000
40000
50000
60000
Размер


Сравнение за времето за изпълнение на алгоритмите за различен обем данни (Отношение време/количество, Сървър, 16 нишки за паралелните)
Разделяне на блокове
Алгоритъм за odd-even merge sort
Адаптивен
80000
60000
40000
20000
0
10000
20000
30000
40000
50000
60000
Размер


Ускорение на алгоритъма спрямо типа на данните (Разделяне по блокове, Компютър, 65536 елемента)
char
int
double
3
2
1
0
5
10
15
20
25
30
Брой нишки


Ускорение на алгоритъма спрямо типа на данните (odd-even merge sort, Компютър, 65536 елемента)
char
int
double
3
2
1
0
5
10
15
20
25
30
Брой нишки

Анализ на резултатите

Адаптивен алгоритъм

Адаптивният алгоритъм има свойството да не изпълнява излишни операции над сортиран масив. В изследването данните са случайно генерирани, но все пак алгоритъмът има огромно предимство пред разделянето на блокове. Колкото е по-малък масивът, толкова по-бързо се справя и адаптивното решение и дори се конкурира с паралелно-рекурсивния алгоритъм за odd-even merge sort. За по-големи масиви адаптивният алгоритъм, макар че ще изпълни много по-малко общи сравнения на данни, се явява десетки пъти по-бавен. Това се дължи главно на по-бързото действие на паралелните решения.

Алгоритъм със SIMD операции

За малки количества данни този алгоритъм се справя значително добре и е сред най-бързите. Техническата му характеристика показва, че няма излишни разклонения в кода (if условия) и изпълнява по две сравнения едновременно. В случая е тестван само за тип INT, но от тестовете проличава колко по-бавен е от merge решението, но и колко по-бърз е от многонишковото разделяне на блокове. SIMD операциите не изискват никаква синхронизация, но имат грануларност, която се явява просто обединение на много операции в една за различни данни. Така стигаме до заключението, че SIMD решението, макар и да не се забавя от синхронизации, не може да се сравни с рекурсивен паралелен алгоритъм.

Алгоритъм за разделяне на блокове

Това е основният алгоритъм в този проект и представлява нестандартен начин за синхронизация и паралелизъм. Очаквано, за малки колкичества данни времето за изчакване на нишките помежду си и системните ресурси те да бъдат пуснати в употреба не обуславят ускорението и дори за голям брой нишки решението става още по-бавно. Повратна точка е използването на нишки, пропорционални на големината на данните, тоест наблюдава се скалируемост. Тогава и изчакването една друга не влияе толкова много, и системните изисквания не са сравнимо големи. Алгоритъмът е значително по-бавен от рекурсивния, понеже във всички тестове на архитектура, поддържаща малко на брой нишки и ядра все пак изчакването и синхронизацията влияят силно. Съществено в алгоритъма е неговото свойство да се възползва от наредбата на паметта в кешовете. Понеже се кешират големи количества информация за типове с по-малки размери, то за PC системата и колкото е по-малък типа толкова по-бързо се обработва от алгоритъма. В Server данните няма толкова съществена разлика, очевидно защото и кешът не е с такъв голям размер и значение.

Алгоритъм за odd-even merge sort

В почти всички случаи това е най-бързият алгоритъм. При него синхронизацията е само колкото е и броят нишки, тоест възможно най-малко. Алгоритъмът е типичен пример за добре организирана сортираща мрежа, която да работи като рекурсивен модел. Подобно на другия паралелен алгоритъм е крайно неефективна за малки масиви ако се ползва с голям брой нишки. Трябва да се отбележи, че заради рекурсивната си специфика решенията с твърде малко нишки не могат да работят правилно поради математическата редица от нива на извикване на функции - именно 2, 6, 14, 30. Тоест решението напасва броя нишки с функциите. Малкото етапи на синхронизация и системни изисквания за създаване на нишки обуславят и бързата работа на решението, която е все по-ефективна за все по-голямо количество данни. Технически слабо зависи от типа на данните, дори си проличава, че има пикове за целочислени данни, които са оптимизирани за почти всички архитектури.

Анализ на броя нишки и големината на данните

За дадените тестови системи стигаме до неоспоримото заключение, че броят нишки е съществено важен спрямо обема на данните (скалируемост при използване). Колкото по-малко данни има, да бъдат сортирани, толкова повече трябва да се избягва паралелното решение, докато за голям брой стойности главно търсим междинна точка, при която алгоритъмът се справя най-добре.

Бъдещо развитие

Към момента проектът цели да разгледа някои основни свойства и разновидности на алгоритъма odd-even sort и да анализира поведението им в различна среда. В никакъв случай не се цели да се разгледат всички възможни комбинации от състояния за алгоритмите, както не се цели и подробност на изложението. В бъдеще реално вече получените изводи трябва да бъдат така приложени, че да се получат нови варианти за реализация на алгоритмите, които макар и силно да зависят от средата, в която се прилагат, то се явяват едно от най-добрите решения. Подобна тясна специализация ще помогне да се изследват крайните случаи на употреба и начините за избягването им. Освен това в бъдеще ще се насочи повече внимание над други свойства (технически и абстрактни), които също да подлежат на проучване.

Заключение

След подробно разглеждане на реализираните алгоритми и изграждането на хипотези за тяхното поведение, както и последващото им тестване можем да направим заключение за коректността на твърденията. Експериментите качествено доказват направените предположения и количествено дават прецизни стойности за това в какъв интервал се променят те за съответните системи. Отбелязани са множество зависимости между имплементациите на всяка сортировка и влиянието на данните и хардуерните особености. Всяка от тези зависимости само утвърждава уникалността на всяко решение и как то е създадено да работи оптимално в дадено поле от случаи. Подробното им изучаване се явява единствен сигурен начин те да се използват ефикасно и ефективно.

Източници

[1] Dana Vrajitoru, “The Parallel Bubble Sort”, IUSB Computer & Information Sciences, USA, 08.2018
https://www.cs.iusb.edu/~danav/teach/b424/b424_15_bubblesort.html
[2] Donald Knuth, “The Art of Computer Programming”, “Sorting and Searching”, Addison-Wesley, 1998
[3] H.W. Lang, “Sorting networks: Odd-even transposition sort”, Hochschule Flensburg, Germany, 04.06.2018
ttps://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/networks/oetsen.htm
[4] B. Jan, B. Montrucchio, C. Ragusa, “Fast parallel sorting algorithms on GPUs”, International Journal of Distributed and Parallel Systems, Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy, 2012
http://www.sce.carleton.ca/faculty/wainer/papers/3612ijdps09.pdf
[5] Y. Kumashev, “GPGPU Verification: Correctness of Odd-Even Transposition Sort Algorithm”, University of Twente, Netherlands, 2020
https://essay.utwente.nl/80585/1/Final_Research_Paper.pdf
[6] K. Batcher, “Sorting networks and their applications”, Goodyear Aerospace Corporation, Akron, Ohio, 1968
https://www.cs.kent.edu/~batcher/sort.pdf
[7] P. Singgih, M. Hagan, R. Wainwright, “Parallel Merge-Sort Algorithms On The Hep”, University of Tulsa
[8] T. Cormen, C. Leiserson, R. Rivest. “Sorting Networks”
[9] Intel® Architectures Software Developer Manuals, “Intel® Intrinsics Guide”, Version 3.6.6
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html