Георги Петров Попов

ФН: 81767, Група: I

BubllePipe:

Паралелно изчисление на Bubble sort със статично балансиране и настройваема гранулярност

Изготвен под ръководството на:
Проф. Д-р Васил Цунижев
Ас. Христо Христов

Съдържание

Съдържание

1. Въведение

1.1 Сортиране

Сортирането е един от фундаменталните части на много и различни алгоритми. Известни са няколко стандартни последователни алгоритъма познати на обществеността – сортиране с вмъкване, сортиране с избиране, сортиране по метода на мехурчето, бързо сортиране, сортиране с обединяване и тн. Тези алгоритми се различават както по сложността си по памет, така и по времевата си сложност.

Алгоритми	**Най-добър случай**	**Среден случай**	**Най-лош случай**
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))
Quick Sort	O(n log(n))	O(n log(n))	$O(n^2)$

Фиг 1. Времеви сложности при различните последователни алгоритми за сертиране.

1.2 Цел на проекта

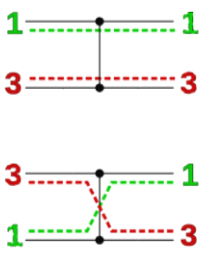
За подобряване на времевата сложност на сортиращите алгоритми са необходими създаването на паралелни техни версии. BubblePipe има за цел да реализира конкретен вариант за паралелна реализация на сортирането по метода на мехурчето. Негова основна цел ще бъде да подобри времето необходимо за сортирането на случаен масив, като се стреми да използва максимално системните ресурси с които разполага. Поведението на сортирането ще бъде разгледано при различни входни параметри – брой елементи, нишки с които разполага програмата, грануларност.

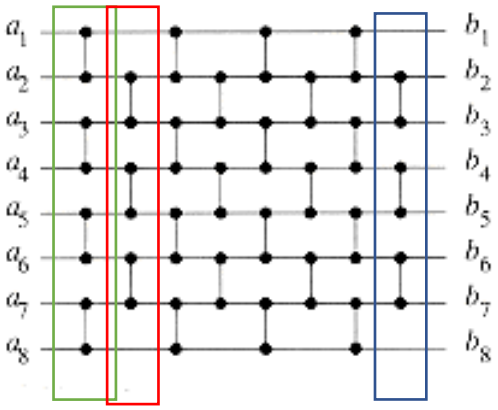
2. Анализ на паралелизма

2.1 Методи за справяне с проблема на последователните сортирания

2.1.1 Паралелно сортиране Odd-Even [1]

Един от стандарните примери за паралелно сортиране е метода Odd-Even. Даже на много места го представят като паралелен Bubble sort. Метода се така наречената сортираща мрежа – при сравнението на два числа изходите може да са два – или да си запазят местата или да се сменят ако първото е по-голямо от второто. Сортировката разглежда масива на четни и нечетни двойки. На всяка итерация – сравняват се всички нечетни двойки или се сравняват всички четни двойки. Като всеки процес заема определен брой четни/нечетни двойки.





ст. 1	ст.2	…	…	ст. n

Фиг2. Стъпки на Odd-Even метода.

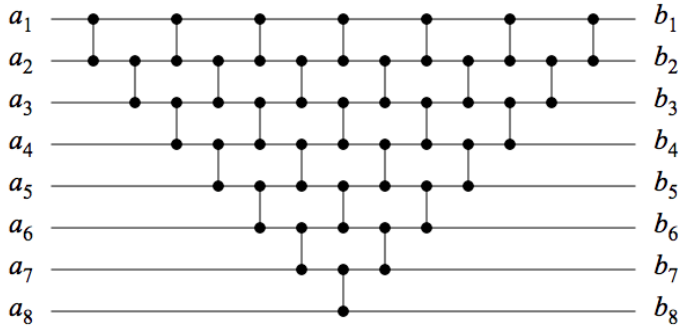
При него са ясно разграничени стъпките какво трябва да се случва на всяка една.

Процес 1	Процес 2	…	Процес t
$a_1 \dots a_k$	$a_{k+1} \dots a_{2k}$		$a_{(n-k+1)} \dots a_t$

Има добра балансировка между работата, която всеки един от процесите извършва. Същевременно и доста лесен за имплементация. Грануларността тук отново може да се адаптира пък размера на L1D-cache – спрямо това колко елемента може да побере и броя на процесите тя се променя – при малък брой процеси – финна – при голям брой преминава към – едра.

2.1.2 Bubble Sort – по метода на сортиращата мрежа

Подобно на горни метод – използвайки сортираща мрежа може и да се предложи вариант наподобяващ метода на мехурчето:

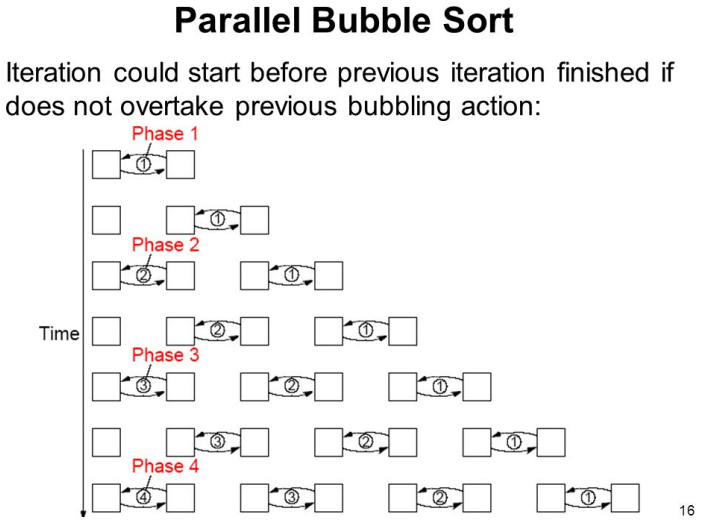


Фиг.3 Odd-Even подход приложен за Bubble sort

Тук обаче веднага се забелязва неравномерно разпределената работа на процесите – докато първия процес ще трябва да извършва много сравнения, то работата на последния е почти нищожна. От тази гледна точко същия подход директно приложен към метода на мехурчето не е много подходящ.

2.1.3 BubblePipe

Ние ще предложим малко по различен вариант. Вместо всеки процес да отговаря за размяната на едни и същи елементи – което е неефикасно поради много по-големия брой нужни смени за началните в сравнение с последните – те ще образуват поточна линия – pipeline – Всеки процес ще обработва даден сегмент от масива и след като приключи работа по него го предава на следващия процес.



Фиг. 4 BubblePipe – идея

По този начин би трябвало ако постоянно увеличаваме броя на нишките, с които разполагаме, да расте и ускорението – ако разбира се разполагаме с достатъчно голям масив

.

Образец	Грануларност	Адаптивност към L1D-cache	Топология	Балансиране
[1] Odd-Even	настойваема	да	SPMD	статично
Bubble – O-E	-	-	SPMD	-
BubblePipe	настойваема	да	MPMD	статично

Фиг5. Сравнителна таблица

3. Проектиране

3.1 Функционално проектиране

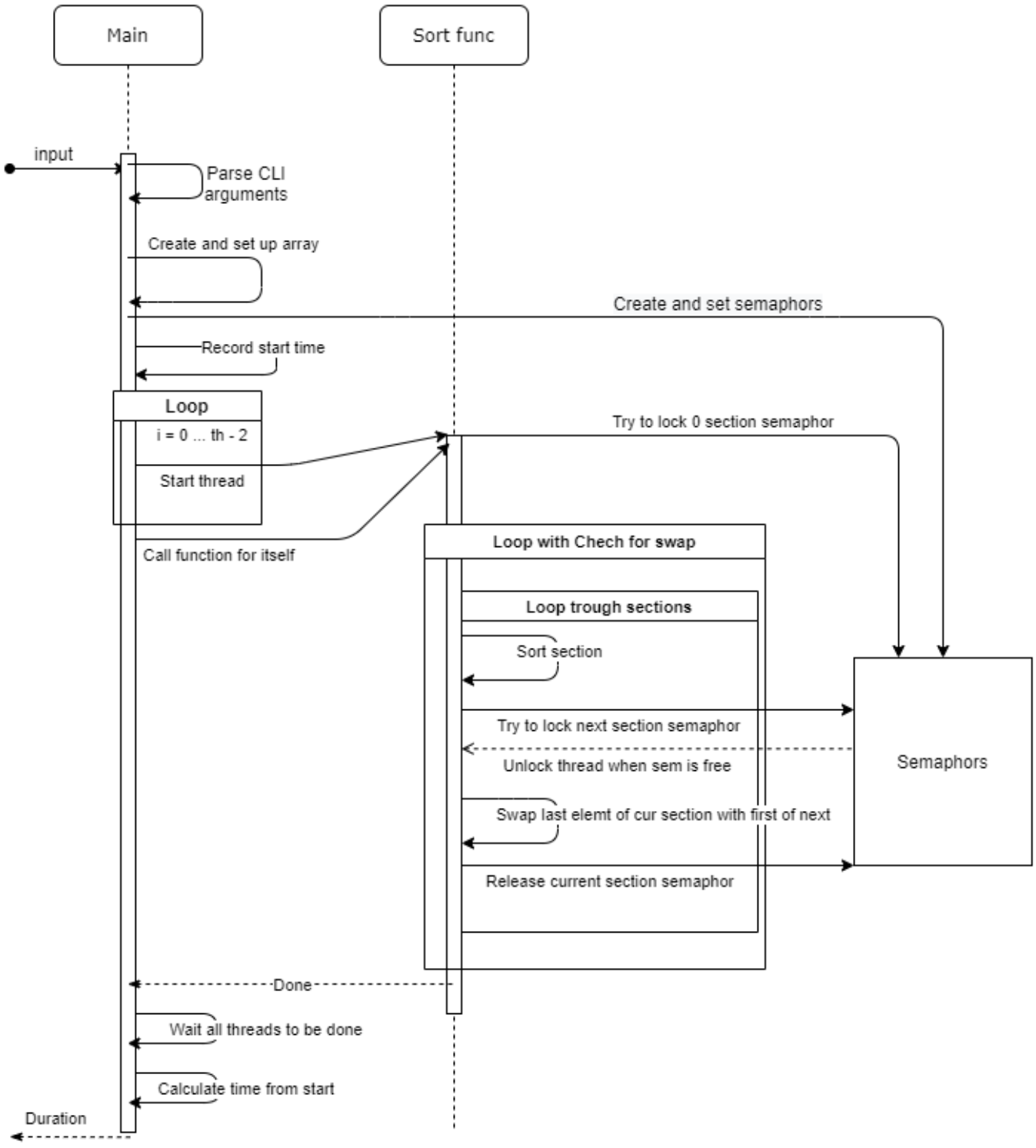
Програмата работи по следния начин – Първо главната нишка създава масива и го запълва със случайни числа. След това го разделя на сегменти, след което за всеки сегмент създава семафор с 1 допустим процес и създава WaitGroup – по който да следи за приключването на работата на нишките. След това задава старта на брояча за изминалото време и започва да създава (th - 1) нишки – където th е броя на нишките – като за всяка и задава да изпълнява функцията, която реално обработва сегментите. След което извиква същата функция и в себе си.

Когато всяка нишка влезне във функцията за обработка на сегментите, първото нещо което прави е да се опита заеме семафора за 0-левия сегмент. По този начин, тъй като семафорите работят по метода на опашката – пръв ще получи достъп първата нишка, след това втората и тн. След това всяка нишка влиза в цикъл, който представлява до кога ще работи, като според номера си, има да работи върху определен брой сектори – като на всяка итерация през цикъла – броя на елементите които трябва да сравни намалява – нишка номер 0 първия път ще обработи всички на първо минаване, след това без последните 1*th - 0, после 2*th - 0 и тн. Нишка номер k при първото си преминаване ще обработи първите n – k елемента, след това 1*th – k и тн.

При влизането в нов сегмент от масива, всяка нишка се опитва да заключи семафора, отговарящ за него. Ако това не е възможно, то тя се приспива и бива събудена кога и дойде реда. Важно нещо е при преминаването на една нишка от един сектор към друг, тъй като трябва да проверява последния от старата секция и първия от новата. Тогава тя преди да отключи сегашния си сегмент, трябва да пробва да заключи следващия. Много трябва да се внимава да не се получи race condition и deadlock.[2],[3] За това трябва да се следи – първо при преминаването от последния сегмент в първия – първо да се отключва и след това заключва – тъй като няма промяна на елементи – второ – да се следи броя на нишките и съотношението им към сегментите – когато станат повече да се намаляват работещите нишки.

След приключване на своята работа, нишките сигнализират WaitGroup - ата, че са приключи и се убиват. Главната нишка след като приключи – се връща в тялото на main-а и чака приключването на всички – ако е свършила преди останалите. След това записва изминалото време от старта и проверява дали масивът е наистина сортиран.

Може да заключим, че реализацията на проекта е с модел MPMD – Pipeline и със статично балансиране. По време на тестовете ще се опитаме да разберем размера на кеша, с който разполагаме за елементи от масива и по този начин да адаптираме грануларността при различния брой нишки и брой елементи в масива.



Фиг 6. UML диаграма на последователността

3.2 Технологично проектиране

Характеристики на машините, на които е тества проекта:

	Build	Home PC	t5600.rmi.yaht.net
CPU	Architecture	x86_64	x86_64
	Threads per core	2	2
	Cores per socket	6	8
	Sockets	1	2
	Total cores	6(12 with hyperthreading)	16 (32 with hyperthreading)
	Model name	AMD Ryzen™ 5 3600 3.60GHz	Intel(R) Xeon E5-2660 @ 2.20GHz
	CPU max MHz	4200	3000
	CPU min MHz	100	1200
	L1d cache	32K	32K
	L1i cache	32K	32K
	L2 cache	512K	256K
	L3 cache	32MB	20480K
RAM	Size	16GB, 3200MHz	62.7GB
OS	Version	Windows 10	CentOS Linux release 7 (core)

Фиг. 7. Характеристики на машините

BubblePipe е имплементиран на Go, език на Google характеризиращ се със своята конкурентна същност и бързо билдване и изпълнение. Много подходящ за системи с паралелна насоченост, още повече когато може да се възползва от вградените му средства за паралелна обработка, както и множеството инструменти и механизми за справяне с проблемите на многозадачните програми, които предлага стандартната библиотека на езика. Същевременно езика е .

Примери от кода:

```
for i := 0; i < *numberOfThreads - 1; i++ {
    wg.Add( delta: 1)
    go serve(semOfSection, &arrayToSort, numberOfElemInSection, numOfSections, i, *numberOfThreads)
```

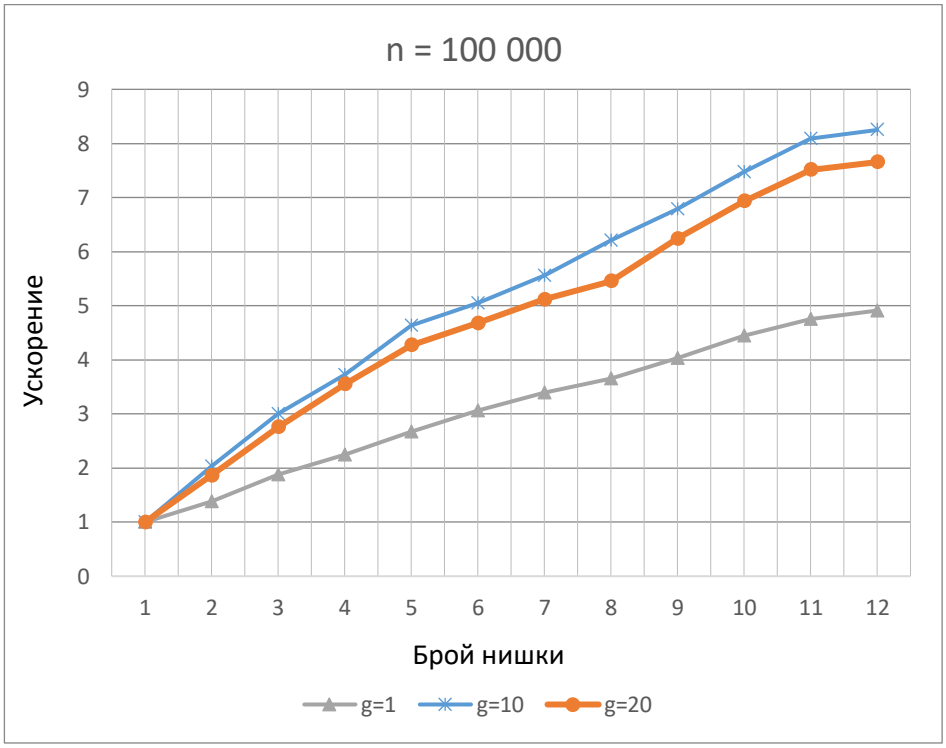
По този начин в GO се създава нова нишка, в която се изпълнява съответната функция, а на горния ред се вижда начина по който се оказва на WaitGroup-а, че ще трябва да се изчака още едно нещо преди да може да се продължи нататъка при извикване на командата wg.Wait()

```
semOfSection[curSection].Release( n: 1)
err := semOfSection[0].Acquire(context.TODO(),  n: 1)
if err != nil {
    log.Fatal(err)
}
```

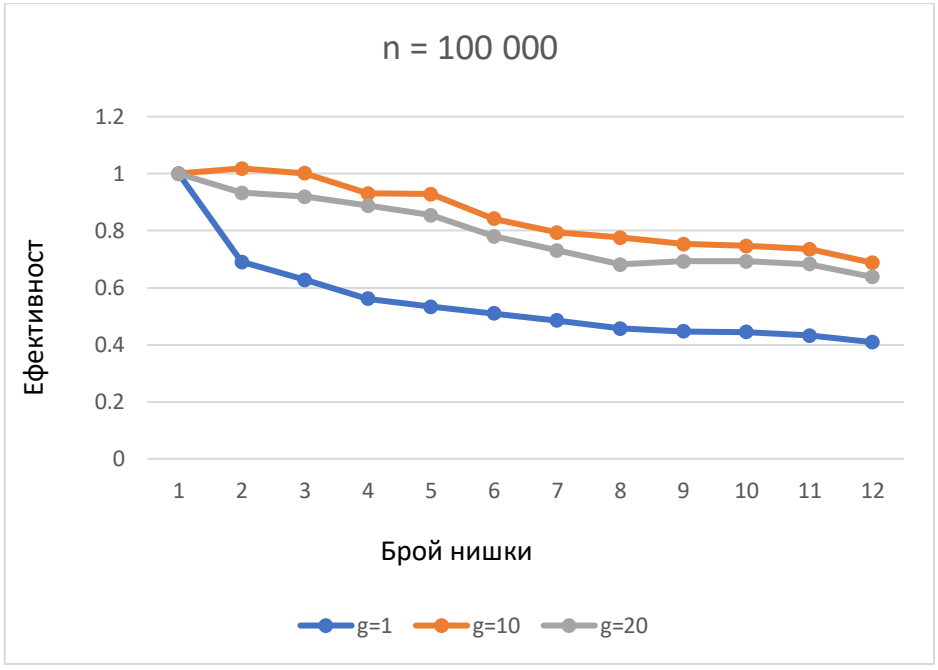
Тук се вижда по какъв начин се освобождава даден семафор за конкретна секция и как се заема за следващата.

4. Тестови резултати

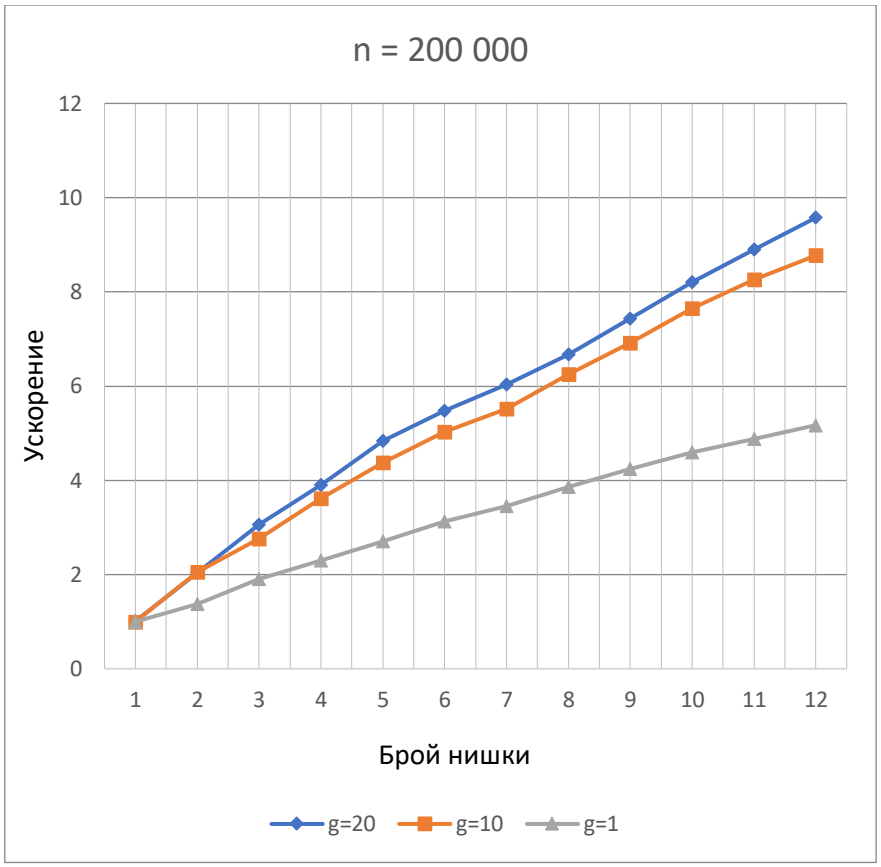
4.1 Home PC



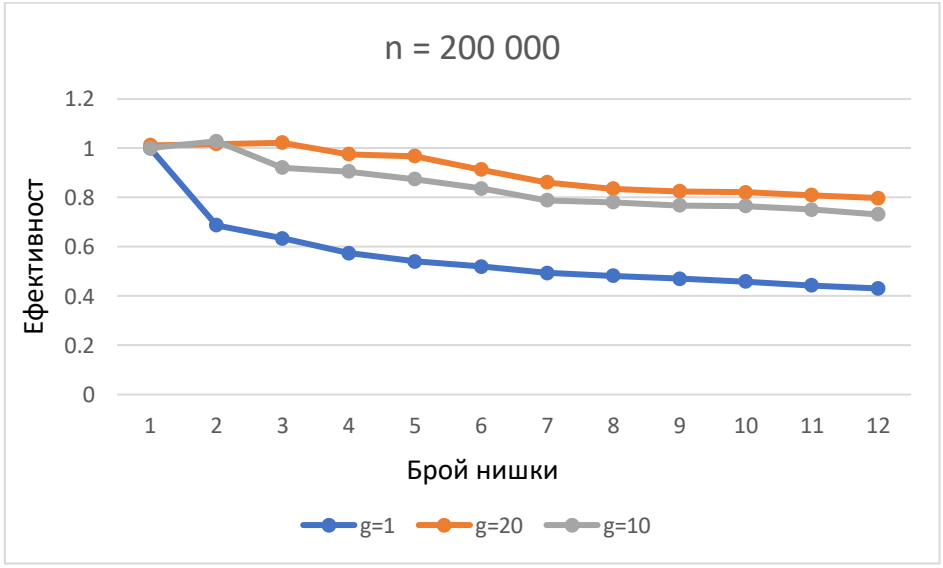
Фиг. 8.1 Ускорение за n = 100 000



Фиг. 8.2 Ефективност за n = 100 000



Фиг. 9.1 Ускорение при n = 200 000



Фиг 9.2 Ефективност при n = 200 000

#	p	G	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	*79 882мс.*	*79 768мс.*	*80 579мс.*	*79 768мс.*	1	1
2	2	1	*57 995мс.*	*58 046мс.*	*58 520мс.*	*57 995мс.*	1.375428916	0.687714458
3	4	1	*34 954мс.*	*34 751мс.*	*34 882мс.*	*34 751мс.*	2.295415959	0.57385399
4	6	1	*25 992мс.*	*25 557мс.*	*25 549мс.*	*25 549мс.*	3.122157423	0.52035957
5	8	1	*21 002мс.*	*20 663мс.*	*20 354мс.*	*20 354мс.*	3.919033114	0.489879139
6	10	1	*17 985мс.*	*17 380мс.*	*18 012мс.*	*17 380мс.*	4.589643268	0.458964327
7	12	1	*15 574мс.*	*15 426мс.*	*16 003мс.*	*15 426мс.*	5.171009983	0.430917499
9	1	10	*76 652мс.*	*75 937мс.*	*78 542мс.*	*75 937мс.*	1	1
10	2	10	*39 941мс.*	*36 929мс.*	*37 542мс.*	*36 929мс.*	2.056297219	1.028148609
11	4	10	*21 741мс.*	*20 964мс.*	*21 113мс.*	*20 964мс.*	3.622257203	0.905564301
12	6	10	*15 264мс.*	*15 117мс.*	*16 440мс.*	*15 117мс.*	5.023285043	2.511642522
13	8	10	*12 345мс.*	*12 154мс.*	*12 274мс.*	*12 154мс.*	6.247901925	0.780987741
14	10	10	*9 965мс.*	*9 921мс.*	*10 100мс.*	*9 921мс.*	7.654167927	0.765416793
15	12	10	*8 546мс.*	*8 657мс.*	*8 754мс.*	*8 546мс.*	8.88567751	0.740473126
17	1	20	*81 952мс.*	*81 045мс.*	*82 000мс.*	*81 045мс.*	1	1
18	2	20	*40 551мс.*	*40 286мс.*	*40 723мс.*	*40 286мс.*	2.011741051	1.005870526
19	4	20	*21 122мс.*	*21 012мс.*	*21 143мс.*	*21 012мс.*	3.857081668	0.964270417
20	6	20	*15 821мс.*	*14 958мс.*	*15 176мс.*	*14 958мс.*	5.418170878	0.90302848
21	8	20	*12 445мс.*	*12 379мс.*	*12 277мс.*	*12 277мс.*	6.601368412	0.825171052
22	10	20	*9 954мс.*	*9 994мс.*	*10 027мс.*	*9 954мс.*	8.141952984	0.814195298
23	10	20	*8 566мс.*	*8 565мс.*	*8 572мс.*	*8 565мс.*	9.46234676	0.946234676

Фиг. 10. Тестова таблица за n = 200 000 в милисекунди

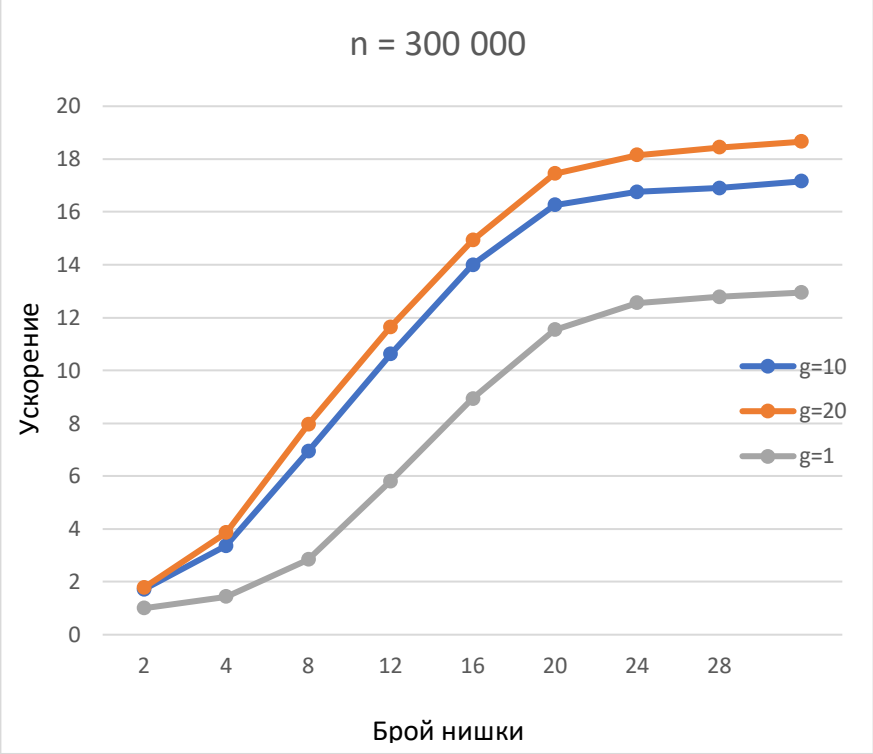
При данните от таблицата, а както и от графиките се забелязва, че и при двата експеримента – при различната грануларност има разлики при ускорението – при 100 000 елемента – има малко по голяма разлика около 6-8 нишки – докато при 200 000 елемента разликата е най голяма при 10-12 нишки. Също така при повечето елементи – се получава по добро ускорение – със статична гранумарност.

Нека разгледаме и един тест с фиксиран размер на блоковете на които разделяме масива – 1666 елемента:

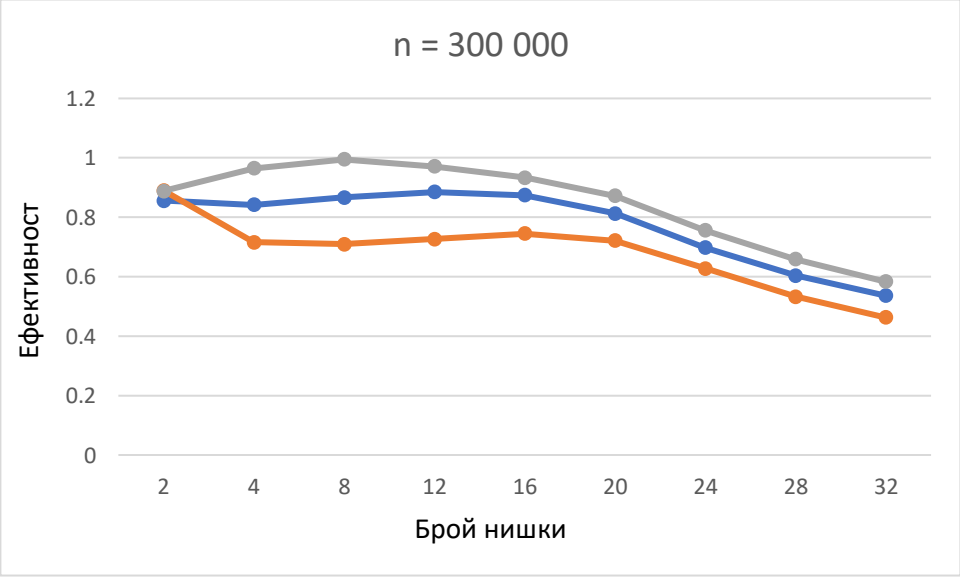
2	37201	8	11155
3	26103	9	10012
4	19991	10	9702
5	16315	11	9053
6	14853	12	8562
7	12725		

Забелязваме, че получаваме по-добри времена от предишните опити – и когато отношението е подобно – са доста близки.

4.2 rmi.yaht.net



Фиг. 11.1 Ускорение за n = 300 000



Фиг. 11.2 Ефективност за n = 300 000

#	p	g	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	*247804*	*239867*	*251402*	239867	1	0.89
2	2	1	*168744*	*167418*	*17012*	167418	1.432743194	0.716371597
3	4	1	*85661*	*88414*	*84571*	84571	2.836279576	0.709069894
4	8	1	*43201*	*41266*	*42007*	41266	5.812702952	0.726587869
5	12	1	*26841*	*28811*	*26994*	26841	8.936589546	0.744715795
6	16	1	*21015*	*20784*	*20884*	20784	11.54094496	0.72130906
7	20	1	*19844*	*19500*	*19114*	19114	12.54928325	0.627464162
8	24	1	*18848*	*18845*	*18755*	18755	12.78949613	0.532895672
9	28	1	*18861*	*18526*	*18667*	18526	12.94758717	0.462413828
10	2	10	*140048*	*14841*	*15452*	140048	1.712748486	0.856374243
11	4	10	*72496*	*71248*	*76201*	71248	3.366648888	0.841662222
12	8	10	*34595*	*34595*	*35897*	34595	6.933574216	0.866696777
13	12	10	*22592*	*23577*	*24456*	22592	10.61734242	0.884778535
14	16	10	*17954*	*17142*	*18624*	17142	13.99294131	0.874558832
15	20	10	*14788*	*14992*	*14751*	14751	16.26106705	0.813053352
16	24	10	*15006*	*14552*	*14320*	14320	16.75048883	0.697937034
17	28	10	*14253*	*14199*	*14191*	14191	16.90275527	0.603669831
18	32	10	*13994*	*13982*	*14011*	13982	17.1554141	0.536106691
19	2	20	*137462*	*135048*	*138144*	135048	1.776161069	0.888080534
20	4	20	*62145*	*62241*	*64001*	62145	3.859795639	0.96494891
21	8	20	*31009*	*31187*	*30148*	30148	7.95631551	0.994539439
22	12	20	*21876*	*22341*	*20592*	20592	11.64855284	0.970712736
23	16	20	*16995*	*16063*	*16120*	16063	14.93288925	0.933305578
24	20	20	*14015*	*13751*	*13954*	13751	17.4436041	0.872180205
25	24	20	*13217*	*13616*	*13307*	13217	18.14836952	0.756182064
26	28	20	*13004*	*13234*	*13601*	13004	18.44563211	0.658772575
27	32	20	*13001*	*12857*	*12879*	12857	18.65652952	0.583016547

Фиг. 12 n = 300 000 в милисекунди

5. Списък с източници

[1] Professor Mohammad Qatawneh, Performance Evaluation Of Parallel Bubble Sort Algorithm On Supercomputer Iman1, Представен: International Journal of Computer Science & Information Technology (IJCSIT) , 2019 (https://www.academia.edu/39787015/PERFORMANCE_EVALUATION_OF_PARALLEL_BUBBLE_SORT_ALGORITHM_ON_SUPERCOMPUTER_IMAN1)

[2] Alan A. A. Donovan · Brian W. Kernighan, The Go programing language, Chapters: 8. Goroutines and Channels and 9. Concurrency with Shared Variables, Публикувана: 26 октомври 2015, ISBN: 978-0134190440 (https://www.gopl.io/)

[3] Dana Vrajitorum, Parallel and Distributed Programming, The Parallel Bubble Sort, (https://www.cs.iusb.edu/~danav/teach/b424/b424_15_bubblesort.html)

[4] Zaid Abdi Alkareem Alyasseri Kadhim Al-Attar, Mazin Nasser, ISMAIL, Parallelize Bubble Sort Algorithm Using OpenMP (https://arxiv.org/ftp/arxiv/papers/1407/1407.6603.pdf)