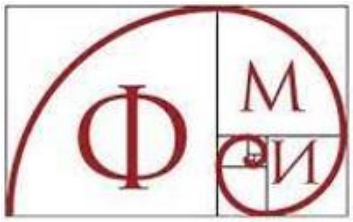
Изследване и сравнение на различните имплементации на паралелен метод на мехурчето

Димитър Кюртов



Компютърни науки, Факултет по Математика и Информатика, Софийски университет „Св. Климент Охридски“

Съдържание

1. Въведение

Сортирането на елементите на структури от данни с произволен достъп на елементи при определена добра наредба е една от най–фундаменталните задачи в програмирането.

Задачата се състои в това да подредим елементите в ненамаляващ или нарастващ ред. Тази задача се среща в почти всяка една съвременна система под някаква форма. Това обяснява защо темата е толкова широко изследвана, в резултат на което са открити доста последователни алгоритми, които решават въпросната задача. Те варират по времева сложност и имплементация. Някои от тях можем да видими в таблица 1 [1].

Таблица 1. Времева сложност на някои последователни сортиращи алгоритми [1].

Sorting Algorithm	Best Case	Average Case	Worst Case
Insertion	O(n)	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Нашият документ ще се съсредоточи върху метода на мехурчето или още наречения бъбъл сорт. По конкретно върху различните начини за имплементиране на паралелна версия на алгоритъма. Ще сравним предложените имплементации на база време за изпълнение, ускорение, грануларност, ефективност на ядрата, цена на синхронизацията и оптимизация на кеша спрямо паралелизма. В целият документ ще приемем, че задачата е да сортираме масива във възходящ ред.

2. Анализ

2.1 Анализ на подобни трудове

Паралелизирането на сортиращи алгоритми е доста богата и изследвана област. Както вече отбелязахме, поради своята популярност има доста голяма нужда от бързи, надеждни и ефикасни алгоритми. В [2] виждаме смесена имплементация между метода на мехурчето и сортировка четни нечетни. В [3] авторите ни предоставят нов начин за използване на паралелен бъбъл сорт за намиране на k най-близки съседа (K-Nearest Neighbour, KNN). Паралелното сортиране може да бъде използвано за подобряване на импементацията на паралелни алгоритми за сигурност като blowfish, RSA, и други [4] [5]. Авторите в [6] разглеждат различни варианти за паралелно сортиране на база разместване на ключовел(data movement), балансиране на

подзадачите (load balance), комуникации (communication latency) и ни дават силна представа за влиянието на различните фактори в производителността на различните сортировки. [7] ни показва възможността за имплементация на бъбъл сорт, използвайки FPGA. [8] ни дава имплементация на паралелен бъбъл сорт, но по-важното е, че фокусира вниманието ни върху най-ефективните стойности на паралелизма, така че да получим максимална ефективност от ядрата и начина, по който да анализираме решението си, така че да достигнем до съответната оптимална стойност.

2.2 Технологичен анализ

Всички имплементации, предложени в този документ, са на написани на езика c++ като стандарта трябва да е c++11 и нагоре. Използвани са само стандартни библиотеки и синхронизационни примитиви. Изборът за технология е такъв, понеже се опитваме да получим максимално добри времеви резултати и език от ниско ниво изглежда най-удачен.

Софтуерните модели, които са използвани са master-slave, data pipelining и divide and conquer. Имаме 2 асинхронни алгоритъма и 2 локално-синхронни като сме използвали декомпозиция по данни (SPMD) и конвейер или декомпозиция по функционалност (MPMD). Използвани са почти всички възможни парадигми с оглед на търсената пълнота на изследване на различните варианти, или по конкренто да видим как всеки фактор влияе върху различните качествени стойности на перформънса на алгоритъма.

3. Потребителски интерфейс

Проектът е достъпен в github [9]. Всяка имплементация има име съответсващо с наименованието и в този документ. Командните аргументи на всички имплементации са еднакви и са подредени в даденият ред:

- Брой на нишки, с които да се изпълнява програмата (естествено число). По подразбиране се извиква с хардуерният паралелизъм.
- Брой на елементите на масива (естествено число). По подразбиране размерът е 40 000.
- Грануларност, брой подзадачи за всяка нишка (естествено число). Тази опция е достъпна само за последната имплементация (BubbleSortMPMD). По подразбиране програмата сама си избира подходяща спрямо останалите входни параметри.

Програмите компилираме със следната команда g++ -pthread -O3 -std=c++11 -o <име на изпълнимия файл, който ще се генерира> <име на програма>

Стартираме с ./<име на програма> <брой нишки> <брой елементи> <грануларност>

4. Проектиране, реализация и тестване

Ще започнем да изследваме различните имплементации в дълбочина. Ще ги разглеждаме във възходящ ред спрямо сложността на имплементацията им.

4.1 Последователен алгоритъм

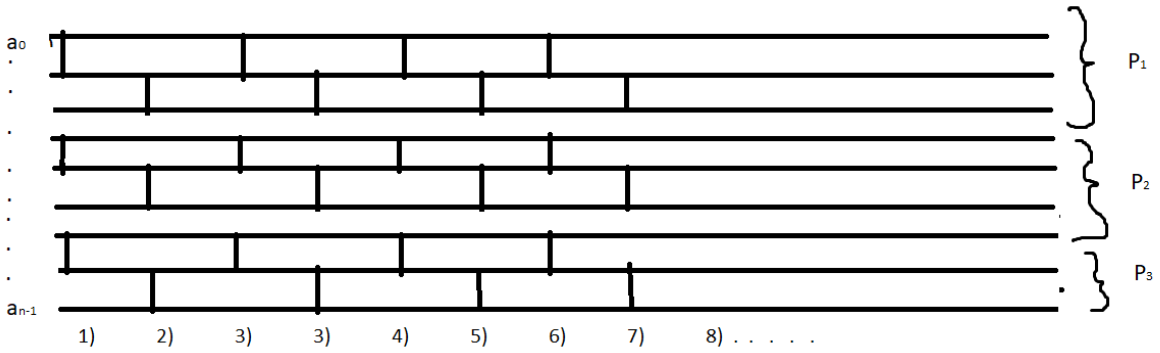
Доброто познаване на последователният алгоритъм е важно за адекватното разбиране на всяка паралелна негова версия. Тези знания не са в обхвата на този документ. В [2] има добро и изчерпателно описание на този алгоритъм. При непознаване на последователният алгоритъм силно се препоръчва старателното му изучаване преди продължаване напред.

4.2 Паралелно мехурче със сливане

4.2.1 Реализация

Имплементацията тук е доста проста, но и доста ефективна в подходящите условия. Декомпозираме масива по данни на равни части. После подаваме всяка част на отделна нишка, която я сортира по отделно, използвайки последователен бъбъл сорт. След като всички подчасти са сортирани, ги съединяваме на принципа: избираме максималният от p максимални елементи на всеки подмасив, където p е паралелизма, с който е пусната програмата.

Фигура 1. Сортираща мрежа на реализацията.



4.2.2 Анализ

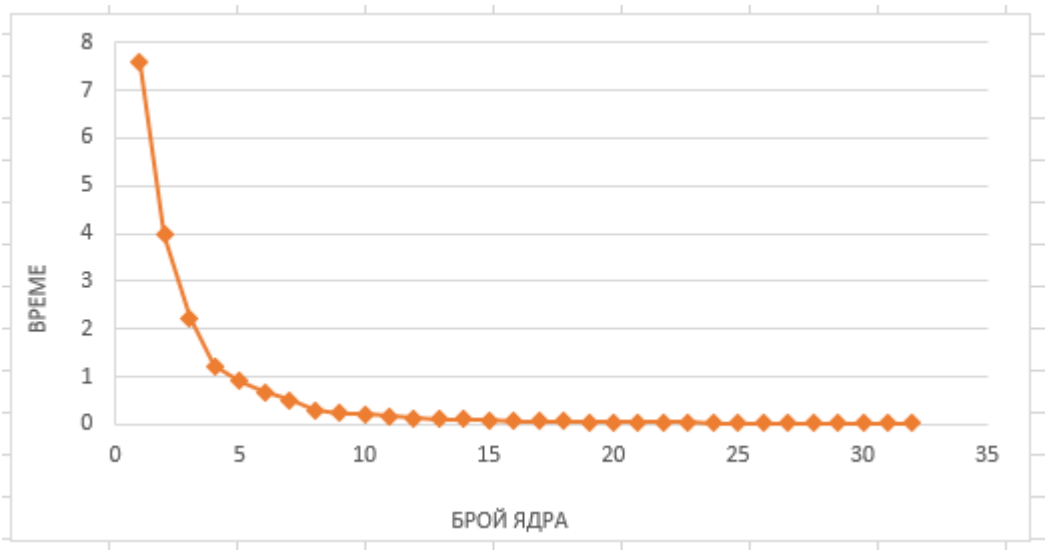
- Цена на комуникацията: почти няма такава, понеже алгоритъмът е асинхронен. Единствено за споделената памет от масивът.
- Баланс на товара: товара на нишките е абсолютно балансиран.
- Грануларност: най-едра, поради добрият баланс.
- Сложност по време: прилагаме последователен бъбъл сорт на p нишки, върху подмасиви с големина n/p. Следователно сложността на тази стъпка е $\Theta(n^2/p^2)$, тъй като всички сортирания се изпълняват в паралел. Сложността на сливането е $\Theta(np)$, тъй като n пъти вадим максималният елемент, имайки избор от p елемента. Така търсената сложност е $\Theta(n^2/p^2 + np)$. Това е хубаво, понеже при малки стойности на паралелизма получаваме квадратично ускорение, понеже превес взима първото събираемо както ще видим при тестването. При голям паралелизъм, при $p = \Theta(n^{1/2})$, използваме приоритетна опашка, за да свалим сложността до $\Theta(n^2/p^2 + n*\log(p)) = \Theta(n*\log(p))$, при съответното p, но това е ненужно и забавя програмата при малки стойности на p.

4.2.3 Тестване

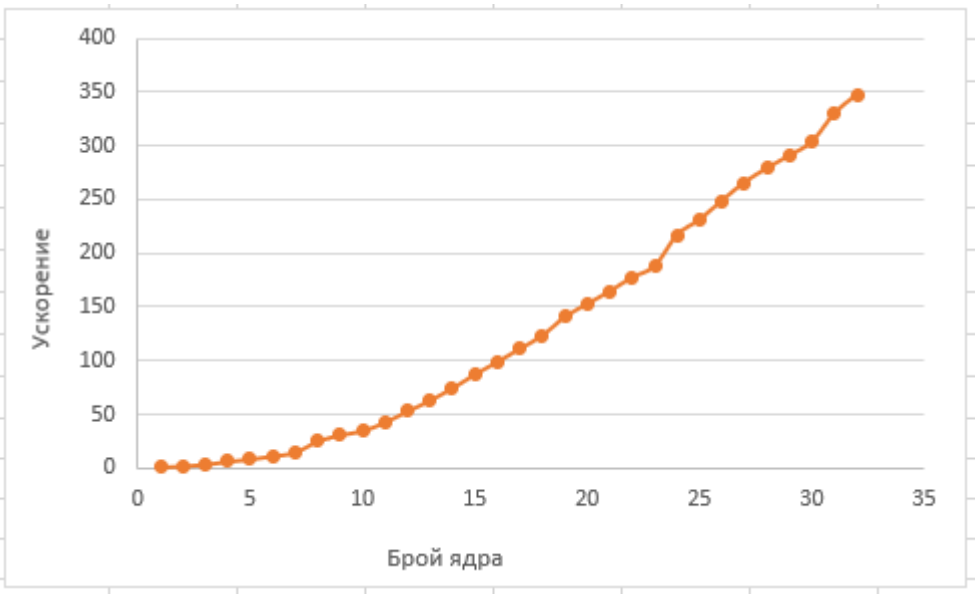
Таблица 2. Измерване на времето на работа, ускорението и ефективността на паралелно мехурче със сливане.

#	p	g	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	$T_p = \min(T_p^{(i)})$	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***7.58146***	***7.59045***	*7.5913*	**7.58146**	1	1
2	2	1	*3.96496*	*3.97274*	*3.97326*	**3.96496**	1.912115129	0.956057564
3	3	1	*2.23484*	*2.24406*	*2.23981*	**2.23484**	3.392394981	1.130798327
4	4	1	*1.21599*	*1.22667*	*1.22158*	**1.21599**	6.234804563	1.558701141
5	5	1	*0.919932*	*0.911745*	*0.917225*	**0.911745**	8.315329396	1.663065879
6	6	1	*0.679746*	*0.676469*	*0.679309*	**0.676469**	11.20740197	1.867900328
7	7	1	*0.52086*	*0.520581*	*0.516546*	**0.516546**	14.67722139	2.096745913
8	8	1	*0.304932*	*0.391982*	*0.388113*	**0.304932**	24.86278908	3.107848635
9	9	1	*0.245859*	*0.305379*	*0.306437*	**0.245859**	30.83661774	3.42629086
10	10	1	*0.247709*	*0.232987*	*0.219202*	**0.219202**	34.58663698	3.458663698
11	11	1	*0.200166*	*0.177995*	*0.180776*	**0.177995**	42.59366836	3.872151669
12	12	1	*0.147049*	*0.145926*	*0.143341*	**0.143341**	52.89107792	4.407589827
13	13	1	*0.120677*	*0.132955*	*0.135653*	**0.120677**	62.82439902	4.832646078
14	14	1	*0.102164*	*0.104432*	*0.102704*	**0.102164**	74.20872323	5.300623088
15	15	1	*0.0883638*	*0.0869194*	*0.08786*	**0.0869194**	87.22402594	5.814935062
16	16	1	*0.076703*	*0.076757*	*0.0768986*	**0.076703**	98.84176629	6.177610393
17	17	1	*0.0683342*	*0.0686063*	*0.0682522*	**0.0682522**	111.0800824	6.534122494
18	18	1	*0.0617414*	*0.0614882*	*0.0618387*	**0.0614882**	123.2994298	6.849968323
19	19	1	*0.0551752*	*0.0538911*	*0.0557883*	**0.0538911**	140.6811143	7.404269175
20	20	1	*0.0500713*	*0.0496436*	*0.0507866*	**0.0496436**	152.7177723	7.635888614
21	21	1	*0.0462862*	*0.0460878*	*0.0465769*	**0.0460878**	164.5003667	7.833350795
22	22	1	*0.0427059*	*0.0427788*	*0.043968*	**0.0427059**	177.5272269	8.069419405
23	23	1	*0.0403359*	*0.0405295*	*0.0413764*	**0.0403359**	187.9581217	8.172092247
24	24	1	*0.0361993*	*0.0363591*	*0.0349533*	**0.0349533**	216.9025528	9.037606368
25	25	1	*0.0332886*	*0.0328024*	*0.0366534*	**0.0328024**	231.1251616	9.245006463
26	26	1	*0.0307748*	*0.0304638*	*0.0315087*	**0.0304638**	248.8678366	9.571839868
27	27	1	*0.0285289*	*0.0287645*	*0.0291571*	**0.0285289**	265.7466639	9.842469034
28	28	1	*0.0273176*	*0.0271842*	*0.027536*	**0.0271842**	278.8921506	9.96043395
29	29	1	*0.0261948*	*0.0260804*	*0.0262066*	**0.0260804**	290.6956949	10.02398948
30	30	1	*0.0249817*	*0.0252194*	*0.0346938*	**0.0249817**	303.4805478	10.11601826
31	31	1	*0.0244378*	*0.0240573*	*0.0229925*	**0.0229925**	329.7362183	10.6366522
32	32	1	*0.0235244*	*0.0235143*	*0.0218248*	**0.0218248**	347.3782119	10.85556912

Фигура 2. Време в секунди за изпълнение при 100 000 елемента на паралелно мехурче със сливане.



Фигура 3. Ускорение при 100 000 елемента на паралелно мехурче със сливане.



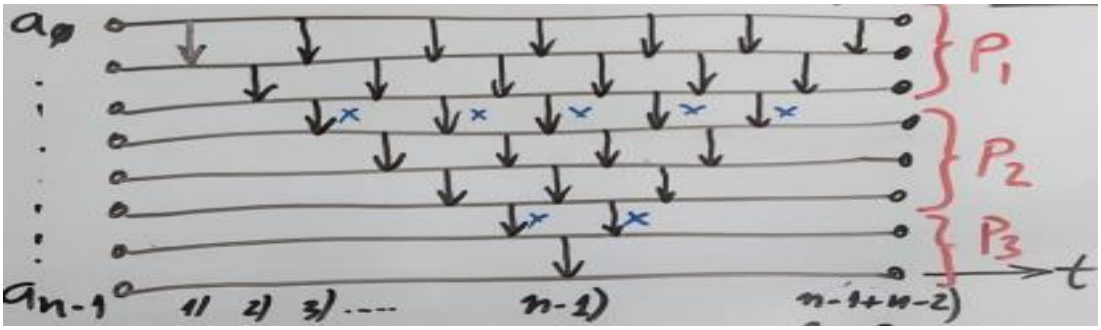
Резултатите съответстват с очакванията ни. Поради малкият паралелизъм получаваме изключително ускорение, но знаем че при по-високи стойности на паралелизма втората фаза ще вземе превес и ускорението ще спадне, но ни трябва машина с по-голям паралелизъм, за да изледваме тези граници. За паралелизъм от този нисък порядък тази имплементация е удачен вариант.

4.3 Паралелно мехурче SPMD със синхронизация

4.3.1 Реализация

В тази имплементация декомпозираме масива на p на брой равни части, където p е паралелизма, с който е пусната програмата. После всяка нишка извършва една итерация на последователното сортиране върху своят подмасив и дава сигнал на следващата, че може да започне. След това всяка си продължава самостоятелно, докато масивът не стане сортиран, което можем да следим, понеже имаме комуникация между процесите (нишките). Единственото ни ограничение е, че не може нишката, която е взела част z да е извършила повече итерации на последователната сортировка от произволна нишка, която е взела част по-малка от z. Ако пък избърза и се опита да го направи се приспива, докато бъде „настигната“ от предишните нишки. Това се прави с оглед коректността на алгоритъма, тъй като за една своя итерация всяка нишка избутва своят максимален елемент от съответният подмасив най-напред и следващата започва като го премества (ако е нужно) в своята област на работа. Така в крайна сметка, когато всички нишки са приключили с първата итерация, максималният елемент в целият масив е на последно място, което се взима предвид от алгоритъма.

Фигура 4. Сортираща мрежа на паралелно мехурче SPMD със синхронизация.



4.3.2 Анализ

- **Цена на комуникацията:** топологията ни предразполага към ниска цена, освен общата памет имаме само примитивни сигнали за приспиване и събуждане.
- **Баланс на товара:** както се вижда от сортиращата мрежа не е балансирано натоварването върху всяка нишка. Това следва от непредвидимостта на входните данни, което води до трудност при опита за баланс на товара.

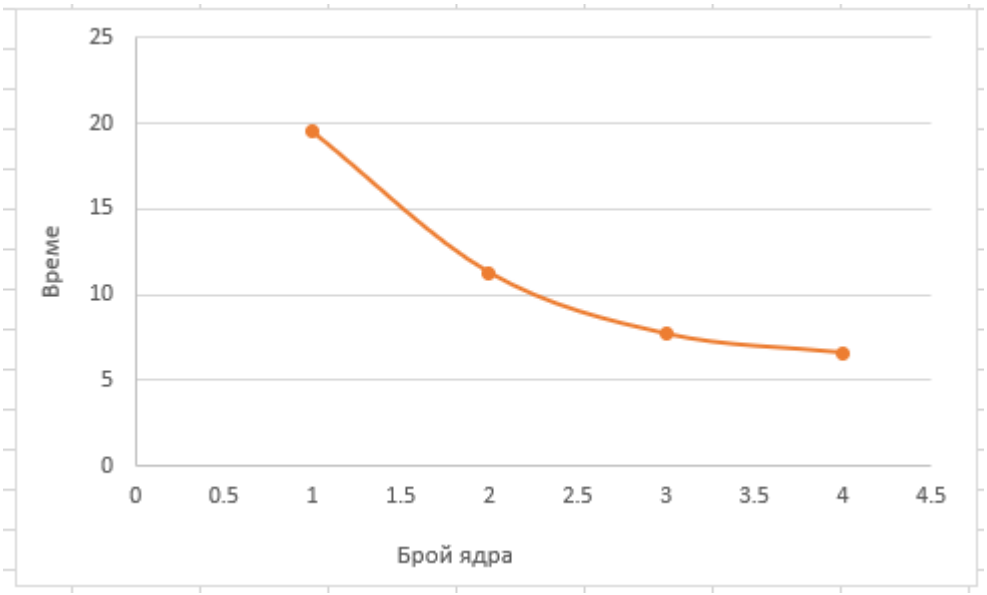
- **Грануларност:** най-едра, поради лошата имплементация (поради желанието ни да проверим ефективността на spmd модела, в следващите имплементации няма такива неточности).
- **Сложност по време:** тъй като всяка итерация на последователният метод се изпълнява паралелно от p нишки и максимумът на броят итерации е n, където n е броят на елементите на масива. Тогава сложността е $\Theta(n^2/p)$, което при достатъчно големи стойности на p може да достигне линейна сложност по броя на елементите на масива.

4.3.3 Тестване

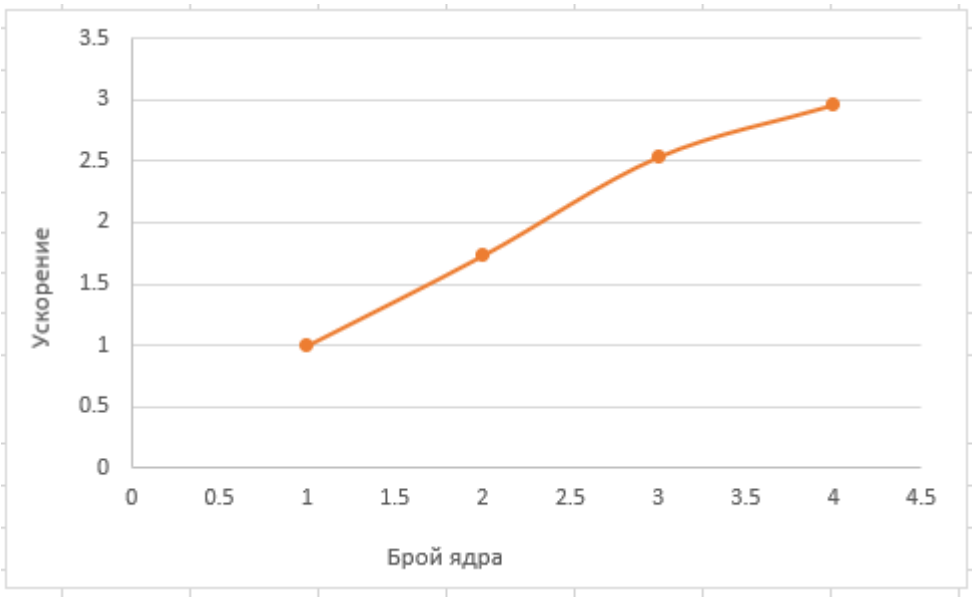
Таблица 3. Измерване на времето на работа, ускорението и ефективността при паралелно мехурче SPMD със синхронизация.

#	p	g	T_p (1)	T_p (2)	T_p (3)	$T_p = \min(T_p^{(i)})$	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	*19.5572*	*19.5198*	*19.5551*	**19.5198**	1	1
2	2	1	*11.4333*	*11.2646*	*11.4158*	**11.2646**	1.732844486	0.866422243
3	3	1	*8.07353*	*7.69736*	*8.21609*	**7.69736**	2.535908415	0.845302805
4	4	1	*6.70706*	*6.59256*	*6.91607*	**6.59256**	2.960883177	0.740220794

Фигура 5. Време в секунди за изпълнение при 60 000 елемента не паралелно мехурче SPMD със синхронизация.



Фигура 6. Ускорение при 60 000 елемента на паралелно мехурче SPMD със синхронизация.



Както предполагахме от анализа, резултатите са меко казано незадоволителни. Поради тези резултати, реализацията не беше тествана на тестовата ни машина. Тестовете бяха проведени на:

- Intel Core i3-8100
- 4 ядра, 3.6 GHz
- L1 cache: 256 KB
- Архитектура x86_64

Тази реализация няма приложим случай, поради по-добрите избори, беше показана, за да видим случай на лошо решение на проблема за паралелизация на алгоритъма. Основната причина за това е лошият баланс на товара. За сравнение ниската ефективност на ядрата, която постигаме тук при 4 ядра (75%), в следващата реализация се достига чак при 18 ядра.

4.4 Паралелно мехурче SPMD без синхронизация

4.4.1 Реализация

Използваме аналогична стратегия като в миналата имплементация. Декомпозираме по данни. Отново всяка нишка се стартира едва след като всички предишни са

приключили със своята първа итерация. Разликата тук е, че нямаме следващи ограничения за синхронизация между тях, освен в точките на синхронизация, за да не развалим интегритета на входният масив. Това очевидно нарушава коректността на алгоритъма, тъй като не е спазен инвариантата от миналата имплементация. Стратегията, която използваме тук е, че задаваме по-голяма горна граница за най-лошият случай. При синхронизираният вариант в най-лошият случай правехме n итерации на целият масив, където n е големината на масива. Тук горната граница пак принадлежи на $\Theta(n)$, но имаме константа отпред, тази константа се изчислява, така че вероятността за неуспех (масива да не се сортира) да е достатъчно малка или по-конкретно 1%. Това се изчислява като взимаме предвид възможността една нишка да изпревари друга достатъчен на брой пъти, повече от този константен множител и да наруши коректността. Като сме приели завишени проценти, за да сме сигурни, че спазваме обещаният процент за успеваемост. Идеята е да проверим колко силно влияе цената на комуникацията върху паралелизма и дали си струва такава. Няма да предоставим сортираща мрежа тук, понеже тя зависи от task scheduler-a. Разбира се при неуспех на сортирането, се извежда подходящо съобщение и се подканва потребителят да пусне отново програмата.

4.4.2 Анализ

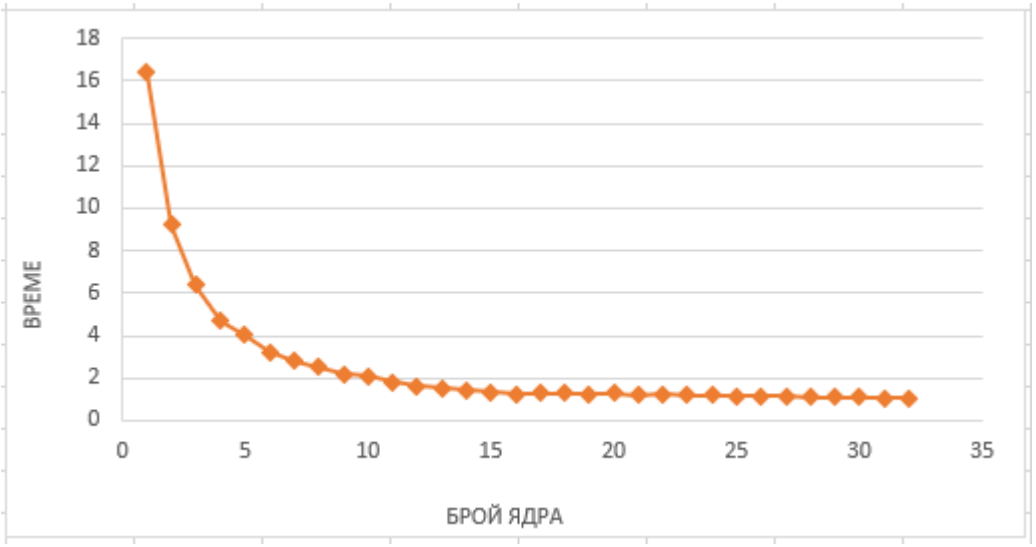
- Цена на комуникацията: пренебрижително малка.
- Баланс на товата: абсолютно балансиран. Всички нишки работят и свършват заедно.
- Грануларност: най-едра, поради добрият баланс.
- Сложност по време: тук имаме c*n итерации на масива, изпълнени в паралел, където c е константата от алгоритъма за вероятност за успех. Следователно общата сложност отново е $\Theta(n^2/p)$, което отново достига линейна сложност при големи стойности на p, въпреки че са еднакви по порядък тук константата пред n е по-голяма, но пък нямаме скрит множител за комуникация или по-точно той е по-малък.

4.4.3 Тестване

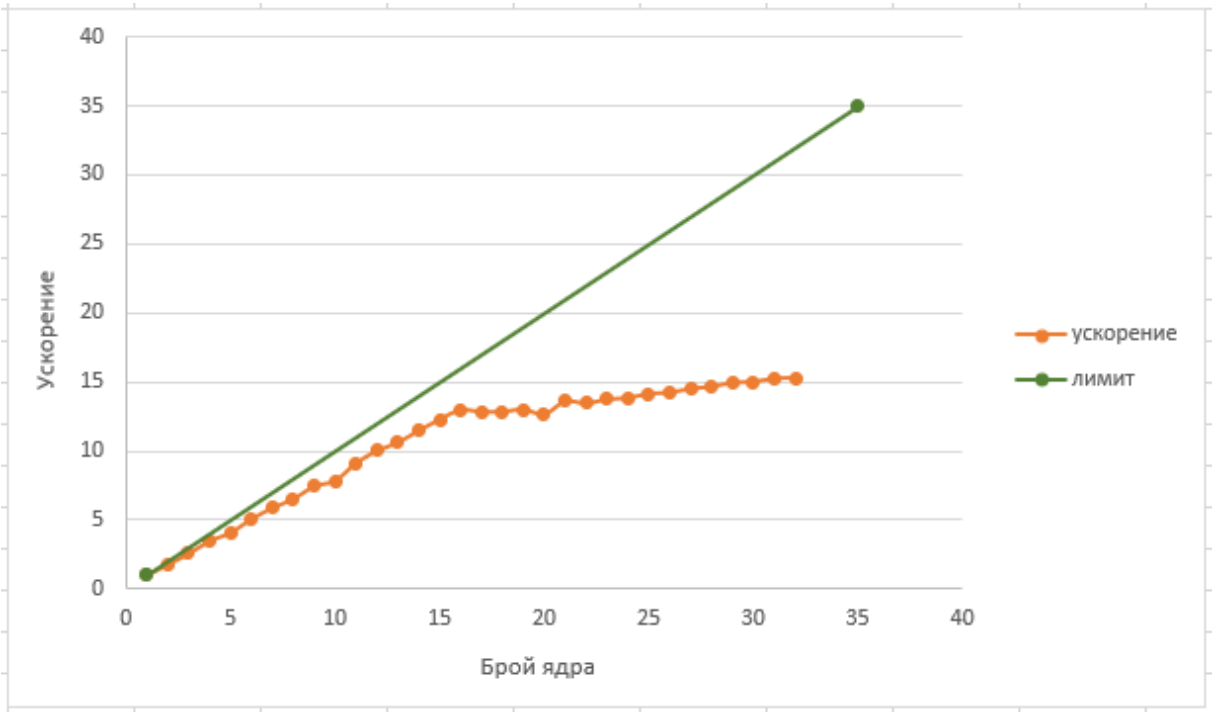
Таблица 4. Измерване на времето на работа, ускорението и ефективността при паралелно мехурче SPMD без синхронизация.

#	p	g	T_p (1)	T_p (2)	T_p (3)	$T_p = \min(T_p(i))$	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***16.4116***	***16.4043***	***16.7557***	**16.4043**	1	1
2	2	1	*9.21508*	*9.20884*	*9.22247*	**9.20884**	1.781364428	0.8906822
3	3	1	*6.38486*	*6.36833*	*6.39167*	**6.36833**	2.575918647	0.8586395
4	4	1	*4.76082*	*4.8373*	*4.70701*	**4.70701**	3.485078638	0.8712697
5	5	1	*4.00927*	*4.01332*	*4.02506*	**4.00927**	4.091592734	0.8183185
6	6	1	*3.22583*	*3.31172*	*3.45098*	**3.22583**	5.085295877	0.8475493
7	7	1	*2.79037*	*2.90042*	*2.90405*	**2.79037**	5.878897781	0.8398425
8	8	1	*2.51158*	*2.54303*	*2.59771*	**2.51158**	6.531466248	0.8164333
9	9	1	*2.20374*	*2.3439*	*2.23322*	**2.20374**	7.443845463	0.8270939
10	10	1	*2.09164*	*2.08925*	*2.10388*	**2.08925**	7.851764987	0.7851765
11	11	1	*1.92428*	*1.80065*	*1.7969*	**1.7969**	9.12922255	0.8299293
12	12	1	*1.65096*	*1.67772*	*1.63881*	**1.63881**	10.00988522	0.8341571
13	13	1	*1.5744*	*1.54597*	*1.59293*	**1.54597**	10.61100798	0.8162314
14	14	1	*1.44804*	*1.50413*	*1.42853*	**1.42853**	11.48334302	0.8202388
15	15	1	*1.3348*	*1.37084*	*1.38786*	**1.3348**	12.28970632	0.8193138
16	16	1	*1.26987*	*1.26982*	*1.26531*	**1.26531**	12.96464898	0.8102906
17	17	1	*1.43357*	*1.27912*	*1.34445*	**1.27912**	12.82467634	0.7543927
18	18	1	*1.33234*	*1.40306*	*1.27837*	**1.27837**	12.83220038	0.7129
19	19	1	*1.31094*	*1.30898*	*1.26079*	**1.26079**	13.01112794	0.6847962
20	20	1	*1.31345*	*1.29643*	*1.30904*	**1.29643**	12.6534406	0.632672
21	21	1	*1.26045*	*1.27958*	*1.20676*	**1.20676**	13.59367231	0.6473177
22	22	1	*1.30658*	*1.27881*	*1.22037*	**1.22037**	13.44207085	0.6110032
23	23	1	*1.22926*	*1.23588*	*1.19358*	**1.19358**	13.74377922	0.5975556
24	24	1	*1.21539*	*1.194*	*1.18614*	**1.18614**	13.82998634	0.5762494
25	25	1	*1.19939*	*1.15916*	*1.16965*	**1.15916**	14.15188585	0.5660754
26	26	1	*1.1535*	*1.16262*	*1.17449*	**1.1535**	14.2213264	0.5469741
27	27	1	*1.13027*	*1.14457*	*1.13844*	**1.13027**	14.51361179	0.5375412
28	28	1	*1.11989*	*1.1293*	*1.14279*	**1.11989**	14.64813508	0.5231477
29	29	1	*1.11184*	*1.12183*	*1.09766*	**1.09766**	14.94479165	0.5153376
30	30	1	*1.09333*	*1.10755*	*1.09551*	**1.09333**	15.00397867	0.5001326
31	31	1	*1.09885*	*1.07555*	*1.08373*	**1.07555**	15.2520106	0.4920003
32	32	1	*1.09023*	*1.07302*	*1.0813*	**1.07302**	15.28797227	0.4777491

Фигура 7. Време в секунди за изпълнение при 100 000 елемента на паралелно мехурче SPMD без синхронизация.



Фигура 8. Ускорение при 100 000 елемента на паралелно мехурче SPMD без синхронизация спрямо хардуерният лимит.



Както виждаме този вариант е задоволителен. Ускорението е почти линейно, до 16 ядра, което е физическият им брой на нашата машина и ако сме склонни да правим компромиси с коректността или някои единични (1 на 1000) заявки да се

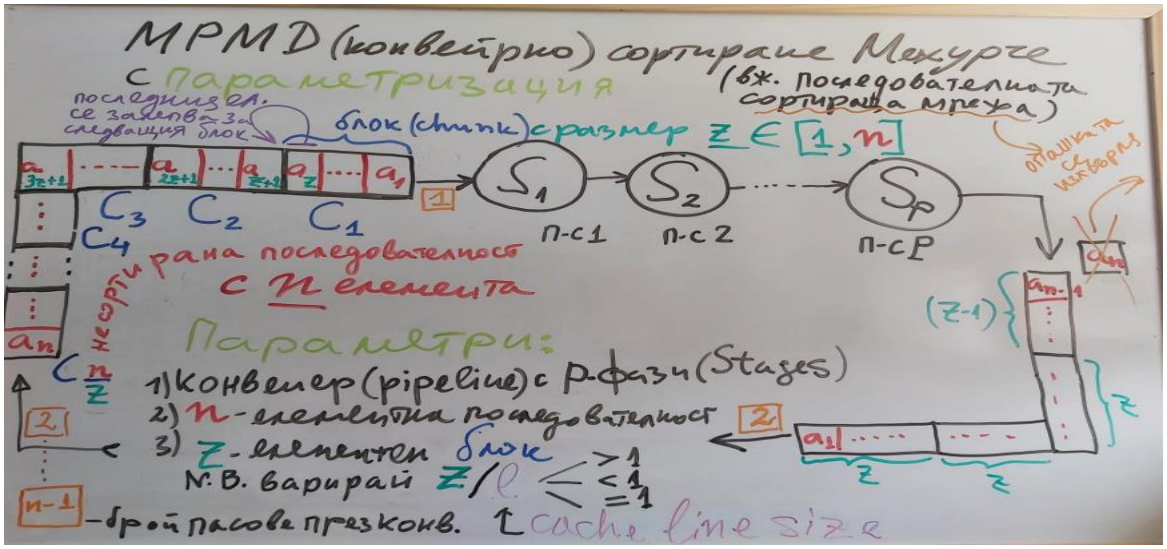
повтарят, когато не се сортират от първият път, този вариант изглежда възможен за употребва. Все пак можем да постигнем и по-добри резултати.

4.5 Паралелно мехурче MPMD

4.5.1 Релизация

Преминаваме към последната имплементация. Ще разгледаме конвейерна имплементация на алгоритъма. Ще разделим масива на оптимален брой части съобразно с размера на кеша и такива, че товара на нишките да е максимално балансиран. Или иначе казано, ще варираме грануларността, така че да получим оптимални стойности за ускорението. Няма да занимаваме читателя с този процес. Той е тестван емпирично и на база оптиамлмите резултати е изграден алгоритъм за намиране на оптимална грануларност. Тук ще дадем готовите резултати и ще обясним защо сме ги избрали пред другите. След като разделим масива на части, всяка част „минава“ през първата нишка, която изпълнява една итерация от последователната сортировка, след това същата част „отива“ при следващата нишка, за да се изпълни следващата итерация от нея, и предишната нишка да може да подхване следващата част за последователно сортиране. Процеса на работа се визуализира със следната схема:

Фигура 7. Схема на конвейерната имплементация.



Както се вижда от схемата всяка част при преминаване на една фаза от конвейера избутва своят максимален елемент на последно място в своят подмасив и вече не го разглеждаме. Тъй като частите влизат последователно, когато всяка част мине през

една фаза, максималният елемент ще е на последно място в целият масив. При достатъчно преминавания на масива през конвейера, той ще се сортира изцяло.

4.5.2 Анализ

В тази имплементация имаме възможност да варираме различните параметри до достигане на оптимални резултати, тук ще ви представим тези оптимални резултати и начинът, по който са избрани

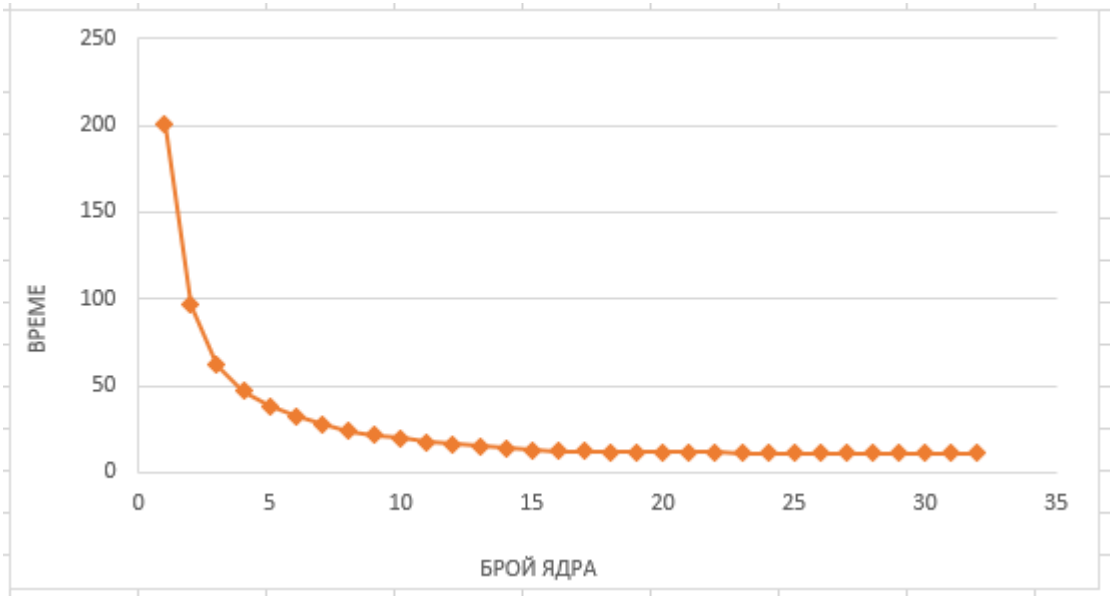
- Цена на комуникацията: тук е възможно най-висока, имайки предвид топологията ни, за да създадем ефекта на конвейер както и да регулираме баланса.
- Баланс на товара: товара е абсолютно балансиран, понеже природата на проблема не предвижда някоя част да е по-трудна от друга, така че факта че всяка нишка обработва еднакво количество информация ни дава увереност за баланса на натоварването.
- Грануларност: варира между едра и средна спрямо броят на нишките и размера на входният масив и размера на кеша.
- Сложност по време: изпълняваме паралелно p последователни сортирания. Следователно сложността отново е $\Theta(n^2/p)$, което при достатъчно големи стойности на p може да достигне линейна сложност по броя на елементите на масива. Разликата тук е, че можем да варираме грануларността, за да оптимизараме другите важни параметри. Това ще видим във фазата на тестване.
- Адаптивност към кеша: при достатъчно много елементи може да се постигне суперлинейно ускорение, поради достигане на размера на L1 кеша като една подзадача. За съжаление това не може да се получи при произволен паралелизъм и прозволен размер на масива, заради нуждата от съвместен баланс на товара заедно с адаптивността към кеша, освен при много много големи стойности на n.

4.5.3 Тестване

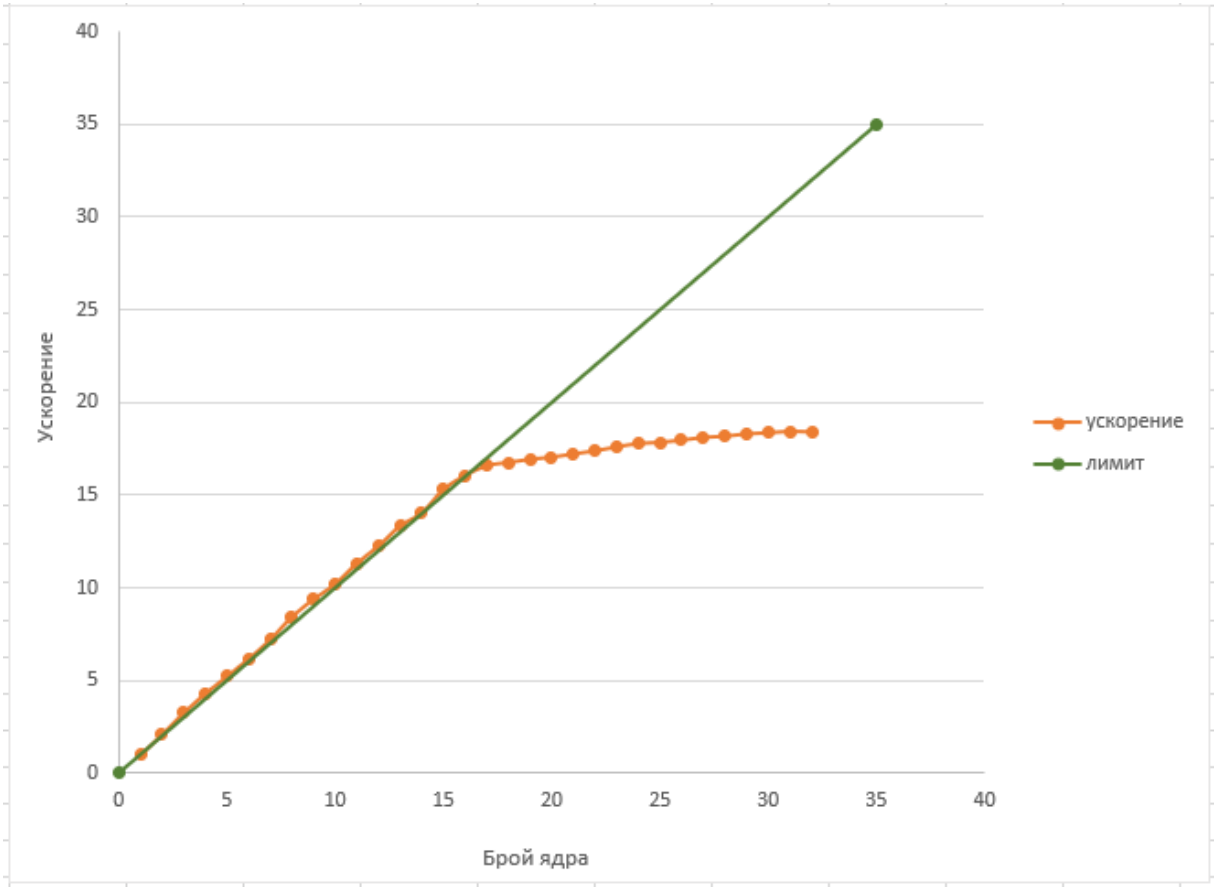
Таблица 5. Измерване на времето на работа, ускорението и ефективността при паралелно мехурче MPMD.

#	p	T_p (1)	48.1753	T_p (3)	$T_p = \min(T_p(i))$	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	201.163	201.081	201.002	201.002	1	1
2	2	104.705	96.6884	102.344	96.6884	2.078863649	1.0394318
3	3	61.6614	62.3658	65.8652	61.6614	3.259770294	1.0865901
4	4	46.7735	47.287	47.1796	46.7735	4.297347857	1.074337
5	5	39.1167	38.5176	38.7722	38.5176	5.218445594	1.0436891
6	6	32.8851	32.5811	32.9518	32.5811	6.169282191	1.0282137
7	7	27.9206	28.0935	27.984	27.9206	7.199057327	1.0284368
8	8	23.8329	23.9366	24.0279	23.8329	8.433803692	1.0542255
9	9	21.4655	21.4239	21.7382	21.4239	9.38213864	1.0424598
10	10	19.7533	20.0869	19.6879	19.6879	10.20941797	1.0209418
11	11	17.9292	17.7621	17.7665	17.7621	11.3163421	1.0287584
12	12	16.402	16.4345	16.4552	16.402	12.25472503	1.0212271
13	13	15.2851	15.2306	15.0758	15.0758	13.33275846	1.0255968
14	14	14.4686	14.4066	14.3158	14.3158	14.04057056	1.0028979
15	15	13.2736	13.299	13.1421	13.1421	15.29451153	1.0196341
16	16	12.5638	12.5232	12.5394	12.5232	16.05037051	1.0031482
17	17	12.3463	12.3221	12.1262	12.1262	16.57584404	0.9750496
18	18	12.0337	12.0617	11.9966	11.9966	16.75491389	0.9308285
19	19	11.9347	11.9561	11.8755	11.8755	16.92577155	0.8908301
20	20	11.8503	11.7981	11.8906	11.7981	17.03681101	0.8518406
21	21	11.6847	11.6935	11.678	11.678	17.21202261	0.8196201
22	22	11.5665	11.6322	11.6333	11.5665	17.37794493	0.7899066
23	23	11.4705	11.4273	11.4392	11.4273	17.58963185	0.7647666
24	24	11.3081	11.3966	11.3988	11.3081	17.77504621	0.7406269
25	25	11.291	11.2705	11.2745	11.2705	17.8343463	0.7133739
26	26	11.1993	11.195	11.2282	11.195	17.9546226	0.6905624
27	27	11.1573	11.1748	11.104	11.104	18.10176513	0.6704357
28	28	11.0796	11.0657	11.0771	11.0657	18.16441798	0.6487292
29	29	11.0593	10.9881	11.0795	10.9881	18.29269846	0.6307827
30	30	10.9535	10.9738	10.9609	10.9535	18.35048158	0.6116827
31	31	10.9151	10.9079	10.9085	10.9079	18.42719497	0.5944256
32	32	10.933	10.9264	10.9309	10.9264	18.39599502	0.5748748

Фигура 9. Време в секунди за изпълнение при 256 000 елемента на паралелно мехурче MPMD.



Фигура 10. Ускорение при 256 000 елемента на паралелно мехурче MPMD спрямо хардуерният лимит.



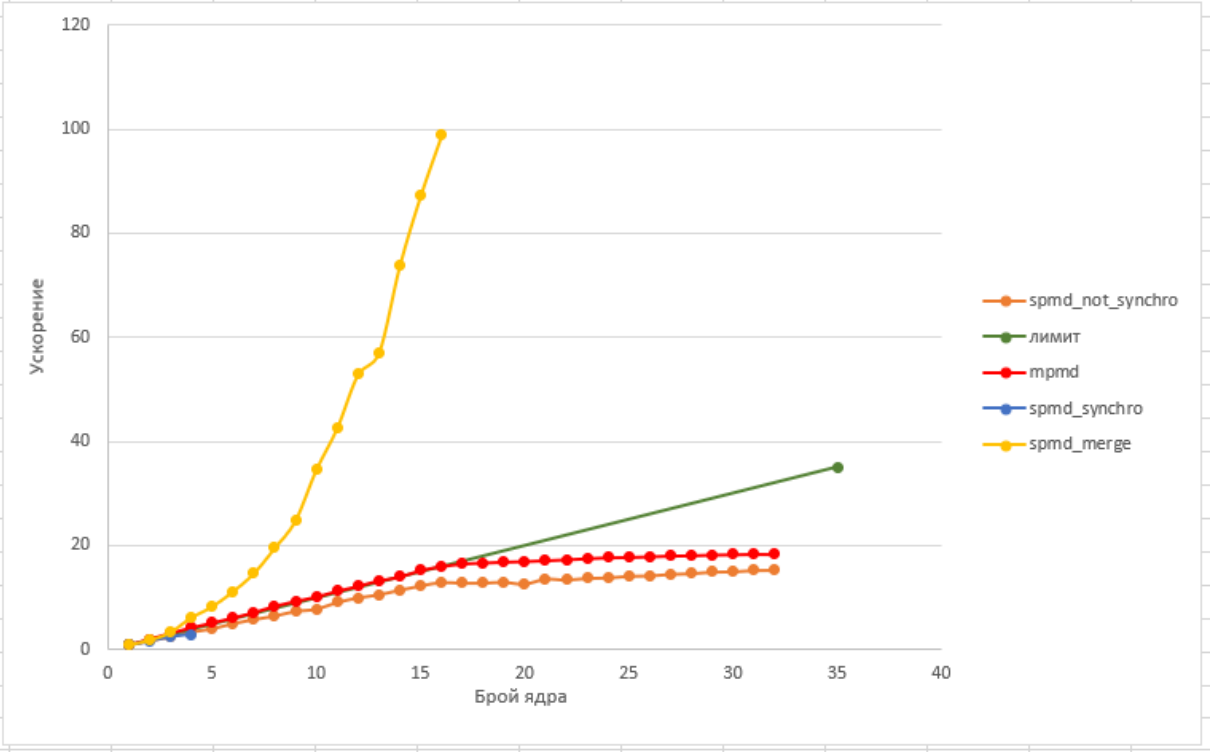
Както очаквахме, тази имплементация дава най-добри резултати. Отново резултатите са много добри до 16 ядра, понеже това е физическият им брой на тестовата ни машина. В таблицата не е показана грануларността поради това, че тя варира спрямо алгоритъма, а не е зададена като команден параметър, ако потребителят държи, може да я зададе като константна.

5. Съвместно тестване на различните имплементации

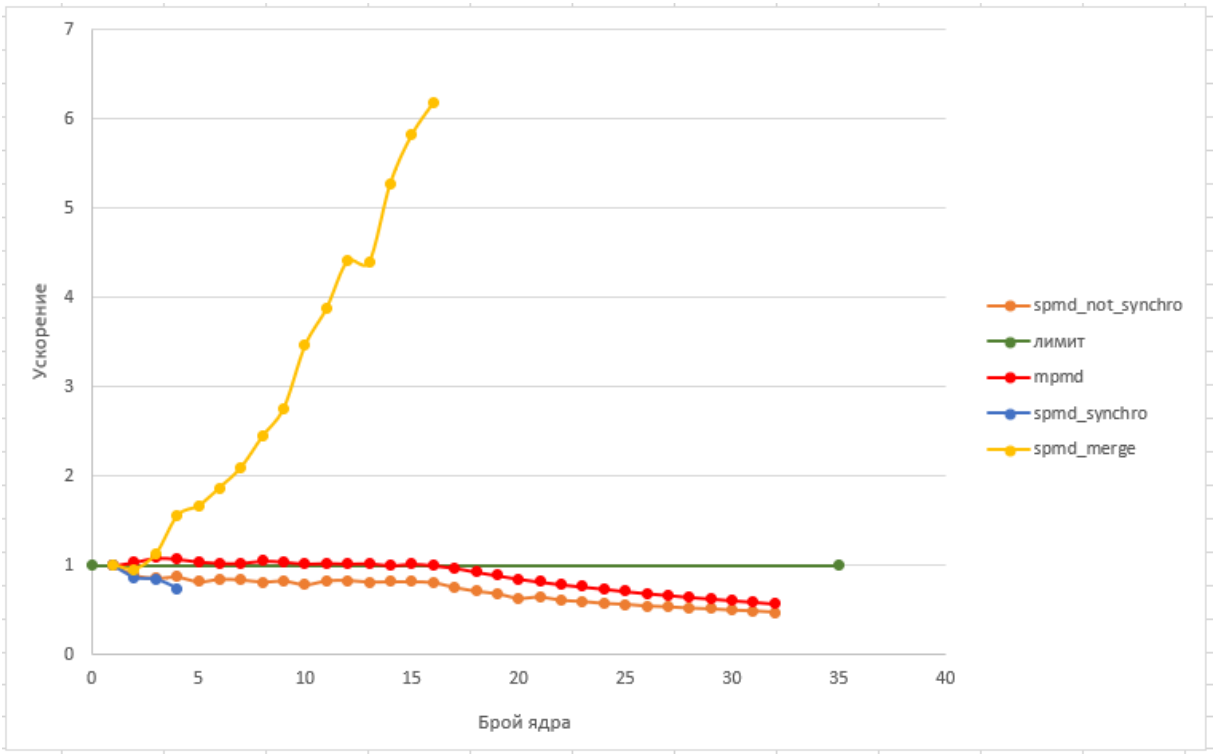
В тази част ще сравним графично всички споменати реализации, за да имаме база да направим финални заключения за ефективността, разликите и приложимите случаи на всяка имплементация. Всички тестове досега и занапред са направени на сървъра на факултета по математика и информатика с изключение на тези, чиито тестова машина е изрично обозначена като друга. Неговите характерисктики са:

- 2 x Intel Xeon @ 2.20 GHz
- 2 x 8 ярда
- L1 cache: 32KB
- Архитектура x86_64

Фигура 11. Съвместно ускорение на всички имплементации.



Фигура 12. Съвместна ефективност на всички имплементации.



Резултатати ни показват, че във общият вариант е най-добре да ползваме конвейерната имплементация, ако обаче знаем, че паралелизмът ни е малък, може да ползваме първата реализация. Другите две са обективно по-лоши във всеки вариант.

6. Източници

[1] A. I. Elnashar, (2011) “Parallel performance of mpi sorting algorithms on dual-core processor windows-based systems,” arXiv preprint arXiv:1105.6040. International Journal of Computer Science & Information Technology (IJCSIT) Vol 11, No 3, June 2019

[2] M. Saadeh, H. Saadeh, and M. Qatawneh, “Performance evaluation of parallel sorting algorithms on iman1 supercomputer,” International Journal of Advanced Science and Technology, vol. 95, pp. 57– 72, 2016.

[3] N. Sismanis, N. Pitsianis, and X. Sun, (2012), “Parallel search of k-nearestneighbors with synchronous operations,” in 2012 IEEE Conference onHigh Performance Extreme Computing. IEEE, 2012, pp. 1–6

[4] M. Asassfeh ,M. Qatawneh, F.AL-Azzeh. (2018), “PERFORMANCE EVALUATION OF BLOWFISH ALGORITHM ON SUPERCOMPUTER IMAN1”. International Journal of Computer Networks & Communications (IJCNC), Vol. 10 (2), 2018

[5] A.Al-Shorman, M. Qatawneh. (2018), “Performance of Parallel RSA on IMAN1 Supercomputer”. International Journal of Computer Applications, Vol. 180 (37)

[6] Vivek Kale, Edgar Solomonik, “Parallel Sorting Pattern”, University of Illinois, Urbana, IL, USA, University of California, Berkeley, CA, USA

[7] Dwi M J Purnomo , Ahmad Arinaldi , Dwi T Priyantini , Ari Wibisono and Andreas Febrian “IMPLEMENTATION OF SERIAL AND PARALLEL BUBBLE SORT ON FPGA”, Faculty of Computer Science, Universitas Indonesia, Kampus Baru UI, Depok, 16424, Indonesi

[8] Sunil Kumar Panigrahi, Sunil Kumar Panigrahi, 2Dr. Soubhik Chakraborty, Dr. Soubhik Chakraborty, 3Dr. jibitesh Mishra, “A Statistical Analysis of Bubble Sort in terms of Serial and Parallel Computation”, Dept of Computer Science and Engineering, APEX Institute of Technology & Management, Pahal, Bhubaneswar-752101, Orissa, India. Department of Applied Mathematics, Birla Institute of Technology, Mesra, Ranchi-835215, Jharkhand, India. Head of Department of Information Technology College of Engg. Technology, Ghatika Bhubaneswar, Orissa, INDIA

[9] https://github.com/dimitarkyurtov/Parallel-Bubble-Sort