Надежда Францева

ФН: 62 391, Група: I

**Parallel Bubble:**

Паралелно изчисление на Bubble sort със статично балансиране и настройваема грануларност



Изготвен под ръководството на:

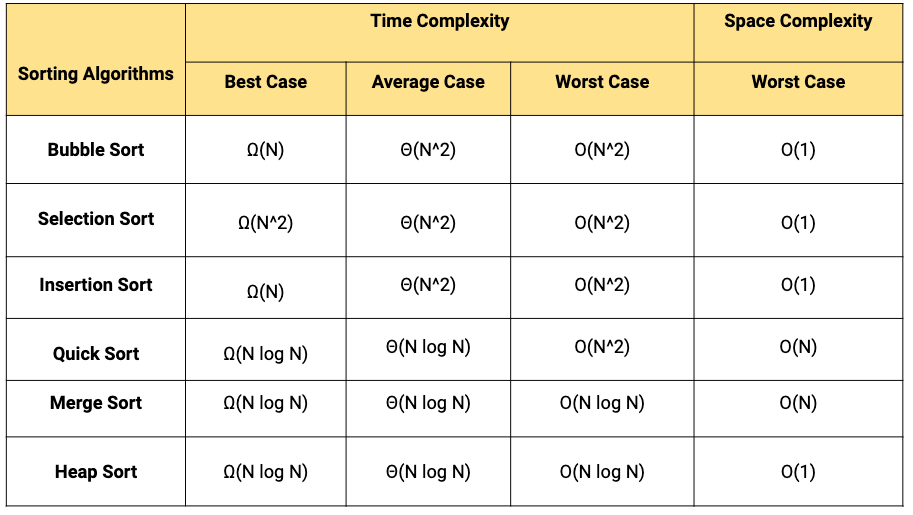
Проф. Д-р Васил Цунижев

Ас. Христо Христов

**1.Въведение**

**1.1 Сортиране**

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



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

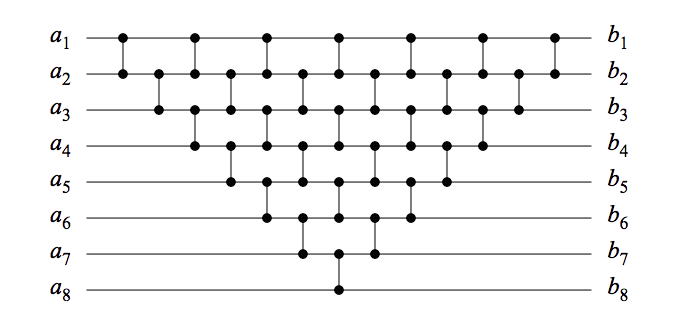
**1.2 Цел на проекта**

За подобряване на времевата сложност на сортиращите алгоритми са необходими създаването на паралелни техни версии. Parallel Bubble има за цел да реализира конкретен вариант за паралелна реализация на сортирането по метода на мехурчето. Негова основна цел ще бъде да **покаже** 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 на масива, нишки с които разполага програмата, **грануларност.**

**2.Анализ на паралелизма**

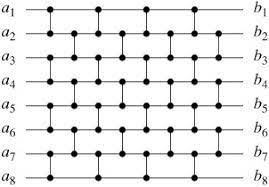
**2.1 Методи за справяне с проблема на последователните сортирания**

**2.1.2 Sorting network of bubble sort**



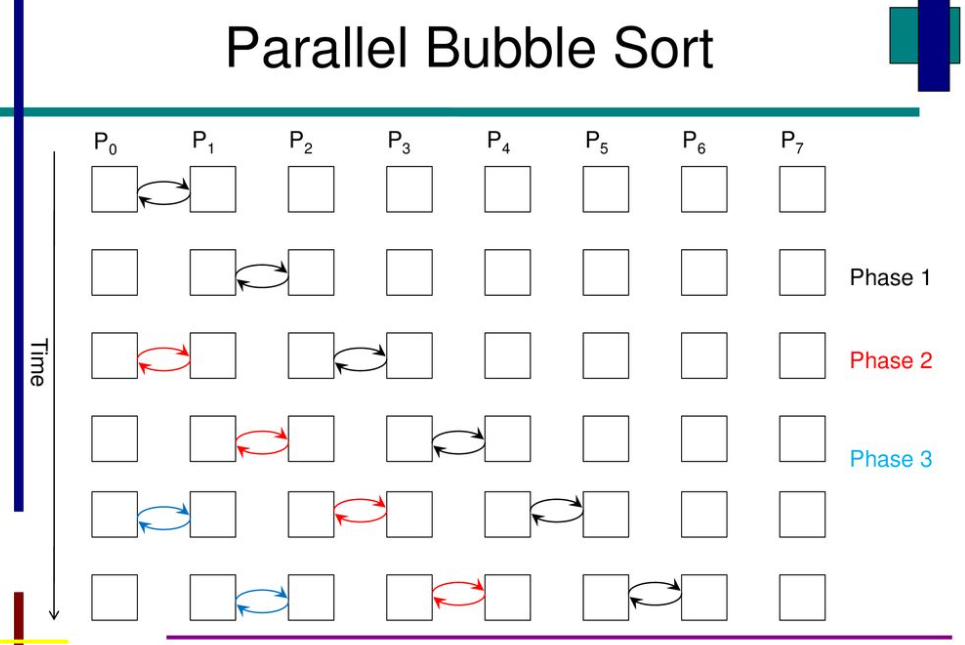
Метода на сортиращата мрежа. Такива мрежи обикновено са проектирани да извършват сортиране на фиксиран брой стойности. Използвайки сортираща мрежа може и да се предложи вариант наподобяващ метода на мехурчето. Тук обаче веднага се забелязва неравномерно разпределената работа на процесите – докато първия процес ще трябва да извършва много сравнения, то работата на последния е почти нищожна. От тази гледна точка същия подход директно приложен към метода на мехурчето не е много подходящ.

**2.1.1 Паралелно сортиране Odd-Even**



Метода Odd-Even е един от стандартните примери за паралелно сортиране. На много места е представен като паралелен Bubble sort. Метода се базира на споменатата 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. При него са ясно разграничени стъпките какво трябва да се случва на всяка една. Има добра балансировка между работата, която всеки един от процесите извършва. Същевременно и доста лесен за имплементация. Грануларността тук отново може да се адаптира към 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.

**2.1.3 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. В началния момент работи само първа нишка, останалите чакат. После се разпространяват по целия конвейер вълнообразно. Всяка нишка ще обработва даден сегмент от масива и след като приключи работа по него минава на следващия, като току-що обработената част я предава на следващата нишка. По този начин, ако разполагаме с достатъчно голям масив и постоянно увеличаваме броя на нишките, с които разполагаме, би трябвало да расте и ускорението.

**3. Проектиране**

**3.1 Функционално проектиране**

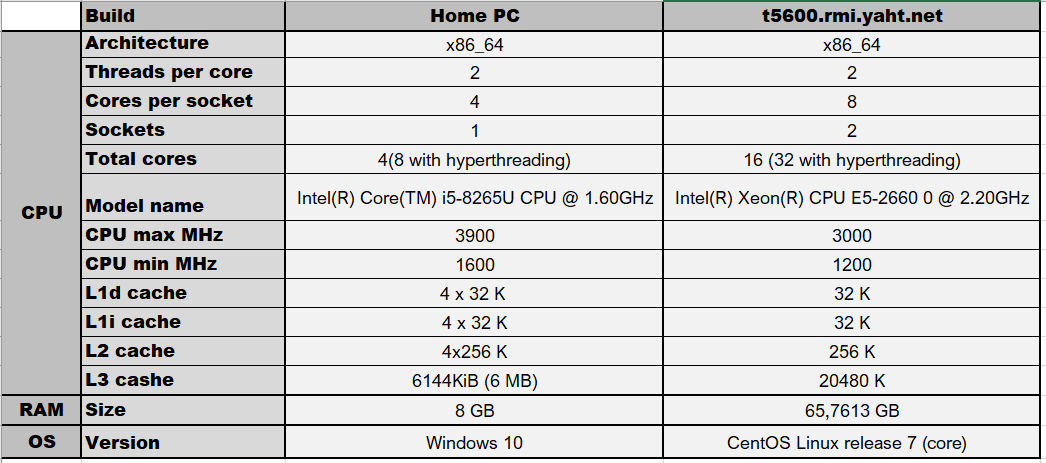
Програмата работи по следния начин:

Първо създаваме масив от критични секции с размер броя на нишките, дадени като аргумент на програмата. Ако броят на елементите на масива се дели без остатък на броя нишки, броят на критичните секции ще бъде равен на броя нишки. Ако се дели с остатък, броят на критичните секции ще бъде равен на броя нишки плюс едно. След това си създаваме масива с размер, подаден като аргумент на програмата и го запълваме със случайни числа.

Следва съществената работа: пускаме нишките от самото начало – всяка нишка започва от критична секция 0. Заключваме сегашната критична секция. Следва for-цикъл с вътрешна итерация, при която разместваме два по два елемента, ако не са на мястото си – в този случай дигаме флаг, че се е случила смяна на числата. Ако сме стигнали до границата на сегашния блок, отключваме критичната секция, в която се намираме и увеличаваме брояча на критичните секции с цел да заемем следващата критична секция. Генерираме следващата точка на освобождаване. След приключването на цикъла, последната нишка освобождава и последната критична секция. Ако имаме смяна в даден блок от масива, ще ни трябва още едно минаване, за да се убедим, дали всичко е сортирано. Ако нямаме смяна никъде, значи целият масив е сортиран. Много трябва да се внимава да не се получи race condition и deadlock. За това трябва да се следи – първо да се отключва и след това заключва и второ – да се следи броя на нишките и съотношението им към сегментите – когато станат повече да се намаляват работещите нишки. Накрая изчакваме всички нишки да приключат работата си и изчисляваме изминалото време от старта. Също проверяваме дали масивът наистина е сортиран. Може да заключим, че реализацията на проекта е с модел SPMD 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.

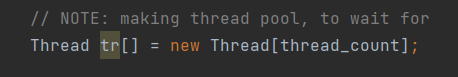
**3.2 Нефункционално проектиране**

**3.2.1 Характеристики на машините, на които е тестван проекта:**

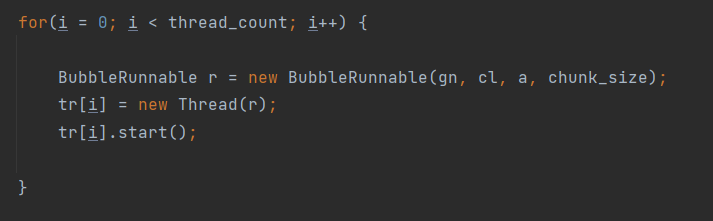
За тестването на програмата бяха използвани следните две тестови среди, като в следващата глава - “4 Тестови резултати” са представени резултатите и от двете тестови машини.

Фиг. ?. Характеристики на машините

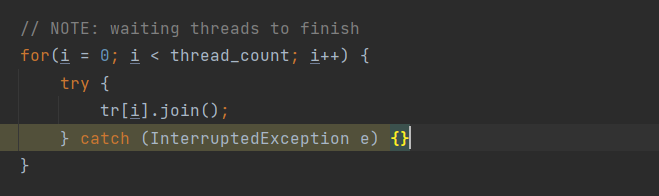
Нашата програма е имплементирана на езика Java. С Java създаването на многонишкови програми е изключително лесно. Достатъчно е да имплементираме интерфейса **Runnable** и да предефинираме метода **public void run()**, в който да напишем програмния код на нашата нишка. При всички положения инстанцирането на класа Thread е задължително, който от своя страна имплементира Runnable интерфейса. След това можем да създаваме обекти от нашия клас и с извикване на метода им **start()** да започнем паралелното изпълнение на написания в тях програмен код. Примери от кода:



Така създадохме масив от нишки.

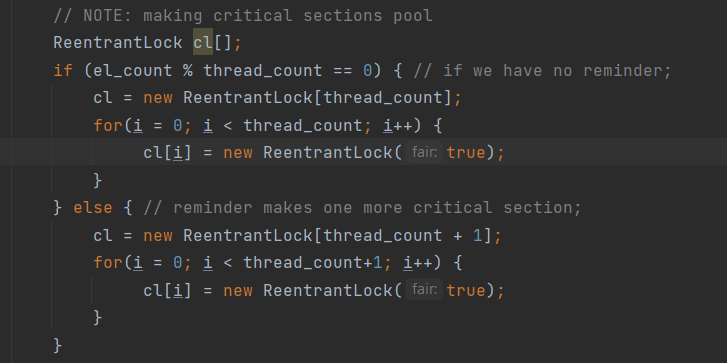


След стартиране на тази програма се създават и стартират <thread\_count> на брой нишки от класа BubbleRunnable.

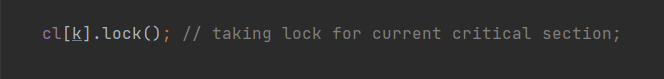


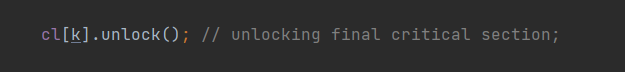
Методът **Thread.join** чака дадената нишка да приключи. Когато извикаме метода **join()** на нишка, извикващата нишка преминава в състояние на изчакване. Остава в това състояние, докато дадената нишка приключи.

Но при многонишковото програмиране е ключово да синхронизираме работата на нишките. Синхронизацията е акт на сериализиране (или подреждане на един по един) на достъпа на нишките до онези последователности на кода, които позволяват на множество нишки да манипулират променливите и други споделени ресурси. Наричаме тези кодови последователности критични кодови секции.



По този начин си създаваме нужния брои критични секции. Класът ReentrantLock реализира интерфейса Lock. А как ги заемаме ?! С методите lock() и unlock() :





**3.2.2 Ръководство за потребителя**

За да може да работи програмата коректно, е необходимо на машината, на която се изпълнява, да има инсталирана Java, версия поне 11, тъй като съм я компилирала с такава версия. Следва списък на параметрите, които програмата поддържа като вход от командния терминал:

Програмата може да се изпълни, като се компилира по следния начин:

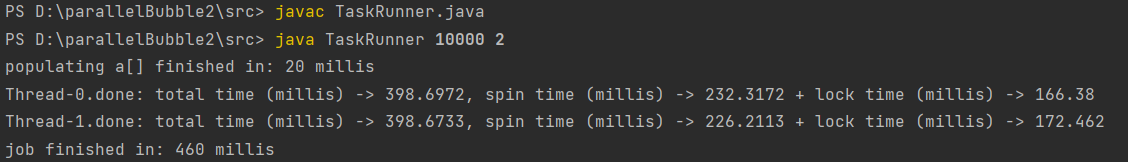
**javac path**

И се изпълни от командния ред по следния начин:

**java path** <**num of elements**> <**num of threads**> **-dump** , където:

1. **num of elements** – брой елементи на масива, който ще се сортира
2. **num of threads** – брой нишки, които се пускат
3. **–dump** – флаг, чрез който искаме да се покажат елементите преди и след сортирането.

Програмата извежда подходящи съобщения на различните етапи от работата си, както и времето отделено за завършване на всяка нишка. Примери за подходящи съобщения:



Където:

**Populating time** – времето за пълненето на масива със случайни числа

**Total time** = spin time + lock time – цялото време на работа на дадената нишка

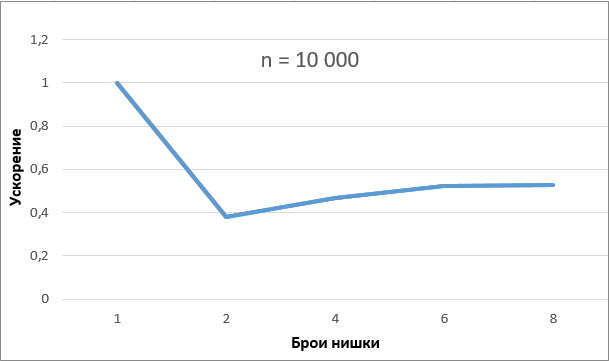
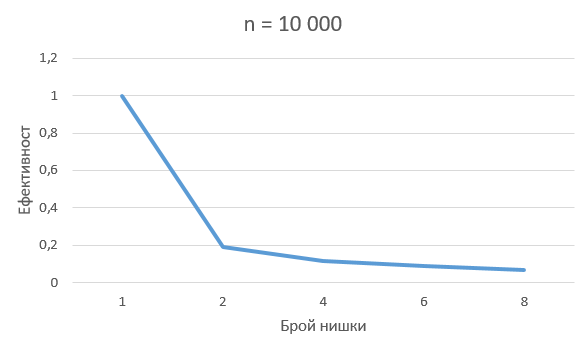
**Spin time** – времето за ходене по елементите от дадена нишка

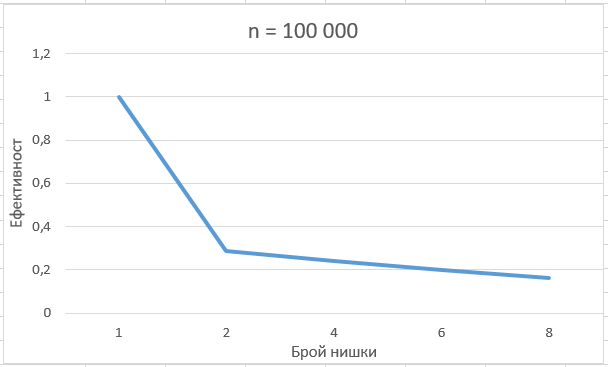
**Lock time** – времето за изчакване да се освободи или да заеме дадена критична секция дадена нишка

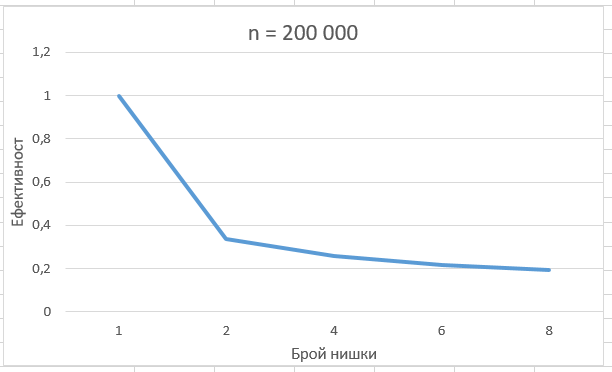
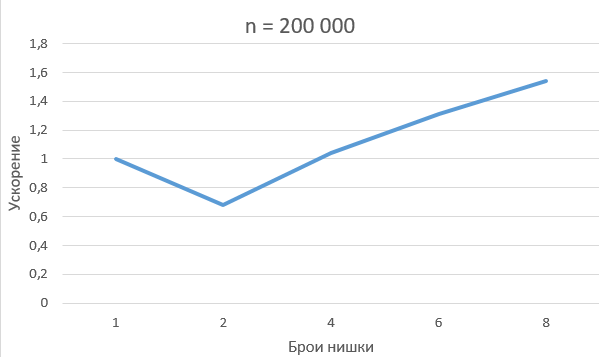
**4.Тестови резултати**

Ще тестваме при настойваема грануларност и ще наблюдаваме резултатите от:

* el\_count - размерът на входния масив
* p - брой нишки
* chunk\_size – размерът на един блок от масива
* Тp (i) - времето за изпъленение на тест i с p нишки
* Tp = min(Tp (i))
* Sp = T1/Tp - ускорение
* Ep = Sp/p – ефективност

**4.1 Home PC**

****

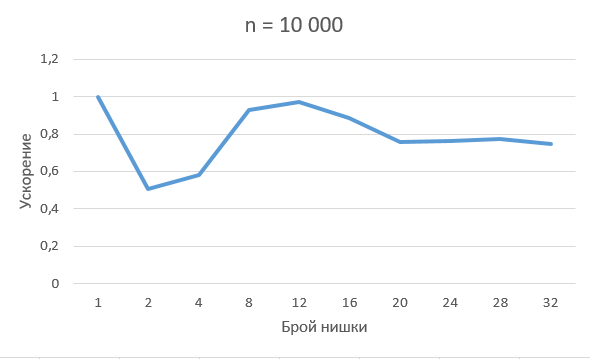
****

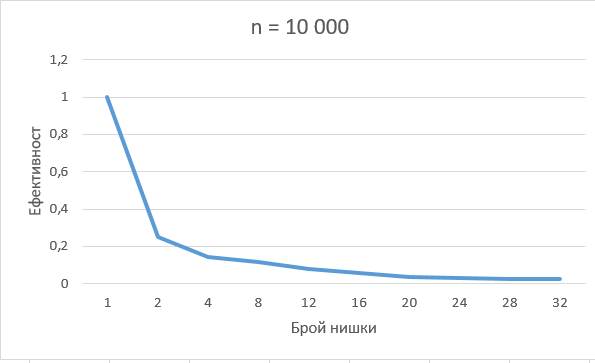
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| el\_count | # | p | chunk\_size | Тр(1) | Тр(2) | Тр(3) | Tp = min() | Sp | Ep |
| n=10000 | 1 | 1 | 10000 | 206 | 185 | 168 | 168 | 1 | 1 |
| 2 | 2 | 5000 | 470 | 440 | 460 | 440 | 0,381818 | 0,190909091 |
| 3 | 4 | 2500 | 412 | 380 | 360 | 360 | 0,466667 | 0,116666667 |
| 4 | 6 | ~1666 | 320 | 406 | 365 | 320 | 0,525 | 0,0875 |
| 5 | 8 | 1250 | 319 | 340 | 371 | 319 | 0,526646 | 0,065830721 |
| n=100000 | 6 | 1 | 100000 | 17162 | 17016 | 20653 | 17016 | 1 | 1 |
| 7 | 2 | 50000 | 31990 | 29934 | 29622 | 29622 | 0,574438 | 0,287218959 |
| 8 | 4 | 25000 | 17521 | 20363 | 19093 | 17521 | 0,971177 | 0,242794361 |
| 9 | 6 | ~16666 | 14321 | 18225 | 17197 | 14321 | 1,188185 | 0,198030864 |
| 10 | 8 | 12500 | 14288 | 16839 | 12899 | 12899 | 1,319172 | 0,164896504 |
| n=200000 | 11 | 1 | 200000 | 90535 | 91507 | 75153 | 75153 | 1 | 1 |
| 12 | 2 | 100000 | 135303 | 110872 | 122012 | 110872 | 0,677836 | 0,338917851 |
| 13 | 4 | 50000 | 87586 | 75959 | 72345 | 72345 | 1,038814 | 0,259703504 |
| 14 | 6 | ~33333 | 62099 | 69178 | 57427 | 57427 | 1,30867 | 0,21811169 |
| 15 | 8 | 25000 | 48817 | 55162 | 54661 | 48817 | 1,539484 | 0,192435525 |

При данните от таблицата, а както и от графиките се забелязва, че и при трите експеримента – при различната големина на входния масив има разлики при ускорението – при 10 000 елемента ускорението страда – докато при повечето елементи – се получава по добро ускорение – със статична грануларност. При 200 000 елемента, поделени на 8 нишки се достига най-високо ускорение ~ 1,5. От данните може да заключим, че за малък масив няма смисъл паралелната обработка, но с увеличаването на големината на масива, се подобрява ускорението поне малко. Стигаме до извода, че тоталната последователната зависимост (Bubble sort) пречи на паралелзма. Не може през цялото време да работят всички процеси, което допълнително влиае. Всеки процес чака през по-голяма част от времето си, вместо да върши работа при малък брой елементи.

Това би могло да се подобри, ако не пускахме всяка нишка от начало, а само в определен блок. Като трябва да внимаваме за две неща: първо – прехвърлянето на данни от един блок в друг и второ – да разберем кога сме сортирали масива. Бихме могли да проверим какво ще стане, ако премахнем lock unlock или друг вариант би бил да разширим този алгоритъм с merge sort. С малка модификация бихме могли да направим и odd-even sort.

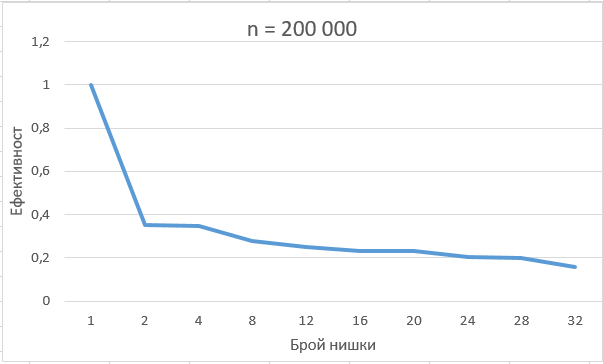
**4.2 rmi.yaht.net**

****

****

****

****

****

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| el\_count | # | p | chunk\_size | Тр(1) | Тр(2) | Тр(3) | Tp = min() | Sp | Ep |
| n = 10 000 | 1 | 1 | 10000 | 277 | 266 | 253 | 253 | 1 | 1 |
| 2 | 2 | 5000 | 642 | 636 | 502 | 502 | 0,503984 | 0,251992032 |
| 3 | 4 | 2500 | 435 | 568 | 508 | 435 | 0,581609 | 0,145402299 |
| 4 | 8 | 1250 | 317 | 273 | 297 | 273 | 0,92674 | 0,115842491 |
| 5 | 12 | ~833 | 260 | 272 | 305 | 260 | 0,973077 | 0,081089744 |
| 6 | 16 | 625 | 294 | 302 | 285 | 285 | 0,887719 | 0,055482456 |
| 7 | 20 | 500 | 334 | 371 | 365 | 334 | 0,757485 | 0,037874251 |
| 8 | 24 | ~416 | 361 | 381 | 331 | 331 | 0,76435 | 0,031847936 |
| 9 | 28 | ~357 | 365 | 326 | 328 | 326 | 0,776074 | 0,027716915 |
| 10 | 32 | ~312 | 354 | 437 | 338 | 338 | 0,748521 | 0,023391272 |
| n = 100 000 | 11 | 1 | 100 000 | 21774 | 21707 | 21679 | 21679 | 1 | 1 |
| 12 | 2 | 50 000 | 40150 | 39267 | 36997 | 36997 | 0,585966 | 0,292983215 |
| 13 | 4 | 25000 | 16153 | 16653 | 18207 | 16153 | 1,342104 | 0,335525908 |
| 14 | 8 | 12500 | 9273 | 9359 | 9345 | 9273 | 2,337863 | 0,292232826 |
| 15 | 12 | ~8333 | 6943 | 6911 | 7031 | 6911 | 3,136883 | 0,261406936 |
| 16 | 16 | 6250 | 5981 | 5773 | 5650 | 5650 | 3,836991 | 0,239811947 |
| 17 | 20 | 5000 | 5251 | 6602 | 6032 | 5251 | 4,128547 | 0,206427347 |
| 18 | 24 | ~4166 | 5390 | 5610 | 5948 | 5390 | 4,022078 | 0,16758658 |
| 19 | 28 | ~3571 | 5286 | 5309 | 4670 | 4670 | 4,642184 | 0,165792291 |
| 20 | 32 | 3125 | 5094 | 5050 | 5056 | 5050 | 4,292871 | 0,134152228 |
| n=200 000 | 21 | 1 | 200000 | 88301 | 87071 | 86824 | 86824 | 1 | 1 |
| 22 | 2 | 100000 | 124502 | 126511 | 123004 | 123004 | 0,705863 | 0,352931612 |
| 23 | 4 | 50000 | 62587 | 63336 | 62265 | 62265 | 1,394427 | 0,348606761 |
| 24 | 8 | 25000 | 39249 | 39172 | 41069 | 39172 | 2,216481 | 0,277060145 |
| 25 | 12 | ~16666 | 28973 | 28843 | 29154 | 28843 | 3,010228 | 0,250852315 |
| 26 | 16 | 12500 | 23411 | 30310 | 24377 | 23411 | 3,708684 | 0,231792747 |
| 27 | 20 | 10000 | 22077 | 18975 | 18691 | 18691 | 4,64523 | 0,232261516 |
| 28 | 24 | ~8333 | 17761 | 19646 | 17649 | 17649 | 4,919486 | 0,204978563 |
| 29 | 28 | ~7142 | 15513 | 17358 | 18618 | 15513 | 5,596854 | 0,199887652 |
| 30 | 32 | 6250 | 18709 | 17743 | 17050 | 17050 | 5,092317 | 0,159134897 |

**5.Списък с източници**

[1] Peter Pacheco – “An Introduction to Parallel Programming” - Morgan Kaufmann Publisher - <http://www.e-tahtam.com/~turgaybilgin/2013-2014-guz/ParalelProgramlama/ParallelProg.pdf>

[2] Cay S. Horstmann – “Core Java Volume I–Fundamentals Eleventh Edition” -<http://rmi.yaht.net/bookz/core.java/11th/9780135170328.pdf>

[3] Kai Hwang, Geoffrey C. Fox, Jack J. Dongarra – “Distributed and Cloud Computing From Parallel Processing to the Internet of Things” - Morgan Kaufmann Publisher -<https://cutepooji.files.wordpress.com/2017/01/distributed-and-cloud-computing-from-parallel-processing-to-the-internet-of-things.pdf>

[4] Presentation on Parallel Computing -<https://slideplayer.com/slide/5053071/>

[5] Article „Parallel Bubble Sort“ - <https://iq.opengenus.org/parallel-bubble-sort/>

[6] Dana Vrajitoru - Article „ The Parallel Bubble Sort“ -<https://www.cs.iusb.edu/~danav/teach/b424/b424_15_bubblesort.html>

[7] Dwi M J Purnomo1, Ahmad Arinaldi1, Dwi T Priyantini1, Ari Wibisono1 and Andreas Febrian2 – “IMPLEMENTATION OF SERIAL AND PARALLEL BUBBLE SORT ON FPGA” - <https://www.researchgate.net/publication/305343153_IMPLEMENTATION_OF_SERIAL_AND_PARALLEL_BUBBLE_SORT_ON_FPGA>

[8] Reem Saadeh and Mohammad Qatawneh - Performance Evaluation Of Parallel Bubble Sort Algorithm On Supercomputer Iman1- International Journal of Computer Science & Information Technology (IJCSIT) , 2019 - <https://www.academia.edu/39787015/PERFORMANCE_EVALUATION_OF_PARALLEL_BUBBLE_SORT_ALGORITHM_ON_SUPERCOMPUTER_IMAN1>

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