**Parallel Bubble Sort**

*(Паралелно изчисление на Bubble Sort - статично балансиране и настройваема грануларност)*

**

Изготвен под ръководството на:

Проф. Д-р Васил Цунижев

Ас. Христо Христов

Изготвил: Божидар Абаджиев

ФН: 9MI0600004, група 4, СИ

Съдържание:

1. Въведение:
   1. Сортирания
      1. Case - worst, average, best
      2. Space complexity
   2. Цел – паралелна версия
2. Анализ
   1. Какво представлява Bubble Sort
   2. Идея, която ще използваме за решение (Odd-Even)
   3. Паралелен Bubble Sort
3. Проектиране
   1. Функционално проектиране
      1. Представяне чрез диаграми
   2. Технологично проектиране
   3. Представено чрез код
4. Тестови резултати
5. Речник
6. Списък с източници
7. Въведение
   1. Сортирания

Нека да започнем като дадем дефиниция на сортиране - Сортирането е процесът на нареждане на елементи от множество в определен ред. Този ред обикновено е възходящ (от по-малки към по-големи) или низходящ (от по-големи към по-малки), в зависимост от изискванията на конкретната задача. Сортирането има широко приложение в програмирането и информатиката, като помага за по-бързо търсене, сравнение и обработка на данни.

Различните видове сортирания имат различни критерии за сравнение. Един от тях е ефективността на алгоритъма - колко бързо може да сортира списъка. Някои алгоритми са по-бързи от други в най-лошите случаи или имат по-голяма степен на ефективност при специфични видове данни. Друг важен критерий е стабилността на сортирането. Стабилен сортиращ алгоритъм запазва реда на равнозначните елементи във входния списък, докато нестабилният алгоритъм не гарантира това. Този аспект е особено важен при сортиране на данни с множество ключове.

Допълнителната памет, която изисква алгоритмът за сортиране, също е важен фактор за сравнение. Някои алгоритми изискват допълнително място за сортиране на данните, докато други работят върху входния списък. И накрая, сложността по време е съществен аспект за оценка на ефективността на алгоритмите за сортиране. Тя показва колко време е необходимо на алгоритъма да сортира входния списък и се измерва в зависимост от броя на сравненията и обмените, които извършва.

* + 1. Сравнение по сложност (worst, average, best case)

В таблицата(фиг 1) представих различни алгоритми за сортиране - bubble sort, selection sort, insertion sort, merge sort, quick sort и heap sort. Всеки от тези алгоритми разполага със свой уникален метод за сортиране на елементите в списъка. За всеки алгоритъм съм посочил времевата сложност в най-лошия, средния и най-добрия случай**.** Изборът на подходящ алгоритъм за конкретна задача често зависи от различни фактори като размерът на входните данни, наличието на допълнителна памет и желаната ефективност.

Фиг 1. Времеви сложности при различните алгоритми за сортиране



* + 1. Space complexity



Фиг 2. Времеви сложности при различните алгоритми за сортиране

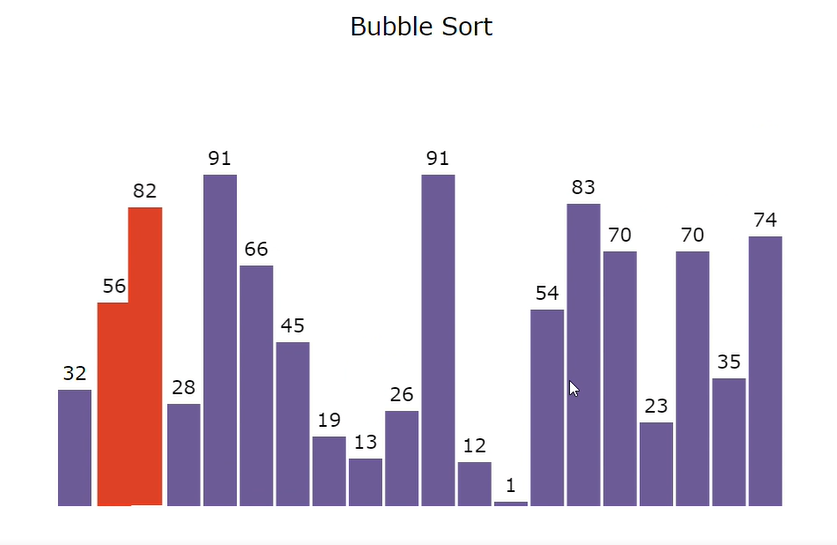
Тук нагледно може да видите и информация за пространствената сложност в най-лошия случай.

* 1. Цел на проекта

Parallel Bubble Sort се стреми към конкретна паралелна реализация на сортирането по метода на мехурчето. Създаването на на паралелно такова сортиране, би подобрило времевата сложност и би използвало същата пространствена сложност(понеже няма да създава нов масив). Главната му цел е да подобри ефективността на сортирането на случаен масив чрез намаляване на времето, необходимо за тази операция. Сортиращият алгоритъм ще бъде тестван, чрез: въвеждане на различни входни данни, тестване на различни устройства, пускане на различен брой нишки и постоянство на връщане на правилни отговори.

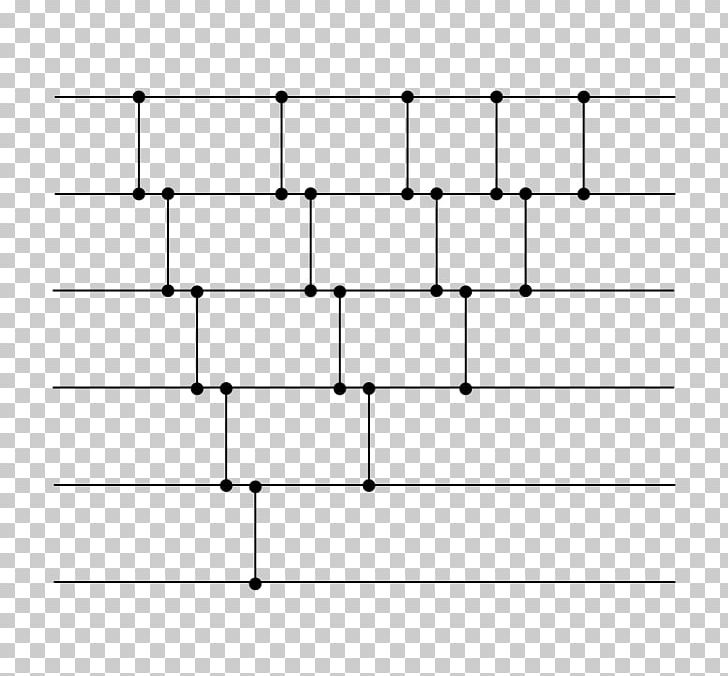
1. Анализ
   1. Представяне на Bubble sort

Алгоритъмът работи като започва от началото на списъка и сравнява всеки два последователни елемента. Ако текущият елемент е по-голям от следващия, те се разменят. Този процес се повтаря над списъка, като всяка итерация "избутва" по-голямите елементи към края на списъка. Този процес продължава докато не се извършат итерации, в които не се правят никакви размени, което означава че списъкът вече е сортиран.

Фиг 3. Изпълнение на Bubble Sort

Както се вижда на фигура 3, елемент 2 и елемент 3 са били разменени като се вижда, че това е първата итерация по започване на сортиране на масива, защото последният не е най-голямото число.

* 1. Разпределение при сортирането



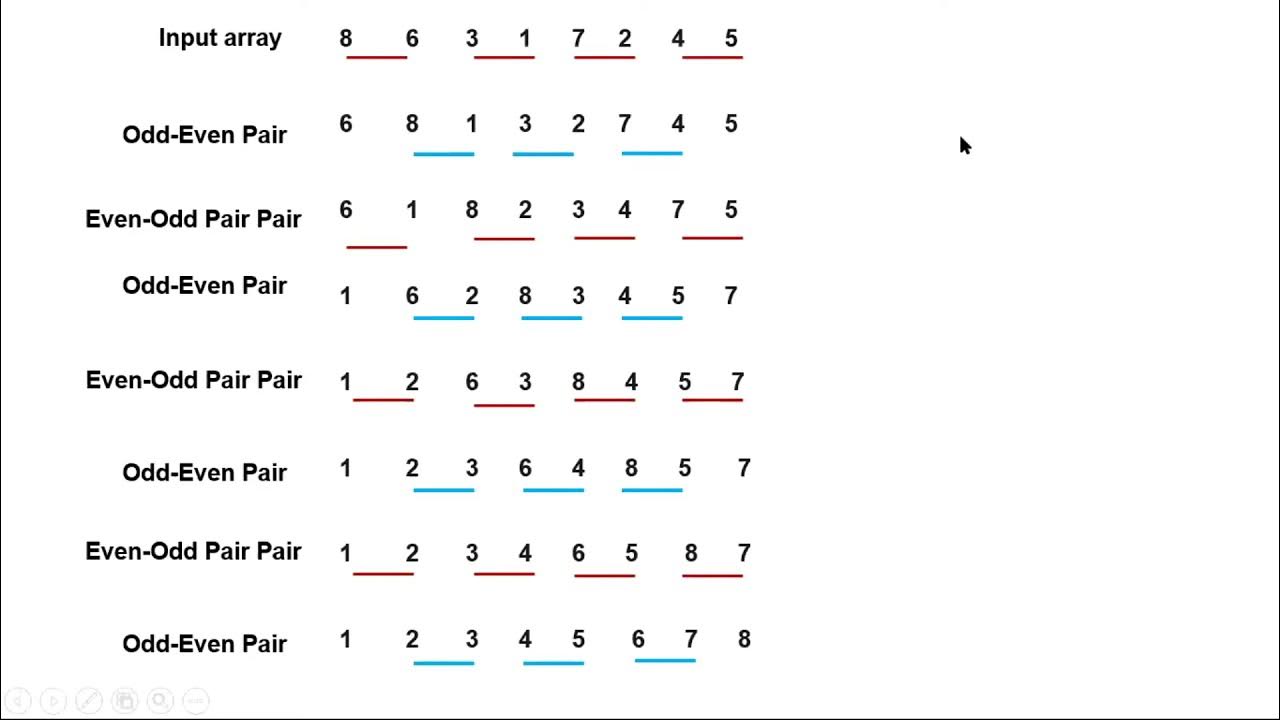
При неизвестен брой елементи една итерация(минаване през всички елементи и сравняването им, даден елемен с неговия следващ) ще се извършва от един процес.

Фиг 4. Сортираща мрежа

Така ако изразим Bubble Sort визуално, ще видим веднага, че се забелязва неравномерно разпределената работа на процесите, докато първия процес ще трябва да извършва много сравнения, то работата на последния е почти нищожна.

* 1. Паралелно сортиране Odd-Even

Един от популярните подходи за паралелно сортиране е методът Odd-Even. Този метод често се свързва с паралелната версия на сортирането по метода на мехурчета. В същността си, Odd-Even методът използва сортираща мрежа, където при сравнение на две числа можем да видим два възможни сценария: числата да запазят позициите си или да се разместят, ако първото е по-голямо от второто. Процесът на сортиране се изпълнява като групираме масива в четни и нечетни двойки. Всяка итерация сравнява всички нечетни или четни двойки, като всяка част от процеса е отговорна за определен брой четни или нечетни двойки.

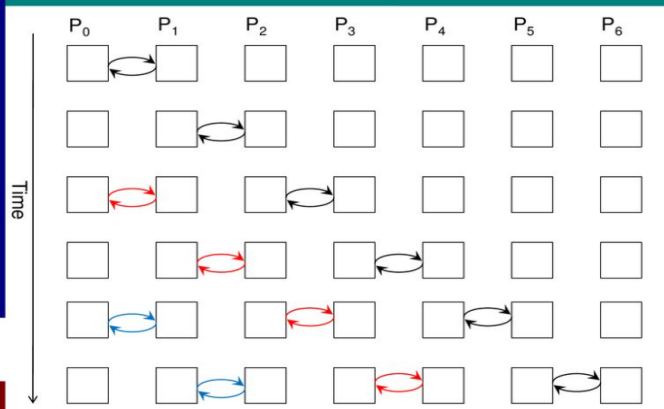
Фиг 4. Изпълнение на Odd-Even Sort 

Този метод е добре структуриран, като всяка стъпка от процеса е ясно дефинирана. Балансът между работата на отделните процеси е оптимален, а методът е сравнително лесен за изпълнение. Гъвкавостта на метода позволява адаптиране към размера на кеша и броя на процесорите. При по-малък брой процеси се предпочита по-фина гранулация, докато при по-голям брой процеси се използва по-едра гранулация.

* 1. Паралелен Bubble Sort

След като разгледахме горните две идеи, стигаме до извода, че Bubble Sort може да бъде оптимизиран, ако създадем комбинирана идея между двете. Стигнахме до извода, че подхода всеки процес да отговаря за проверката и размяната на едни и същи елементи става леко бавен и неефикасен. Забелязва се, че смените се случват в пъти повече в началните елементи отколкото в крайните елементи. Което ни подсказва, че трябва да обработваме елементите постоянно.

Стигаме до идеята, че всека нишка ще обработва дадени два елемента от масива(ел1 и ел2) и след като приключи работа с тях, запазва 2рия, освобождава първия и взима следващият в масива. Като за новата двойка елементи действието по сравняване и размяна се извършва отново. А при освобождаването на елемент1 друга нишка вече може да го вземе и да го използва за ново сравнение и размяна.



Какво очакваме да се случи при развитието на идеята – ускорение при увеличаване на нишките.

1. Проектиране
   1. Функционално проектиране
      1. Представяне чрез диаграми
   2. Технологично проектиране
   3. Представено чрез код
2. Тестови резултати
3. Речник
4. Списък с източници