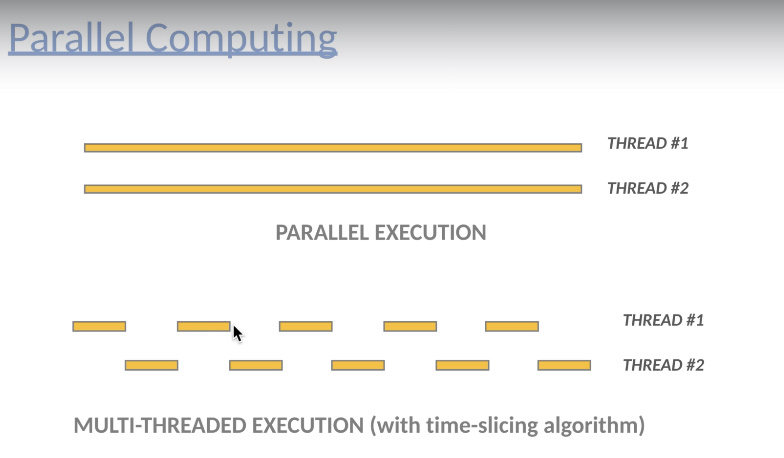
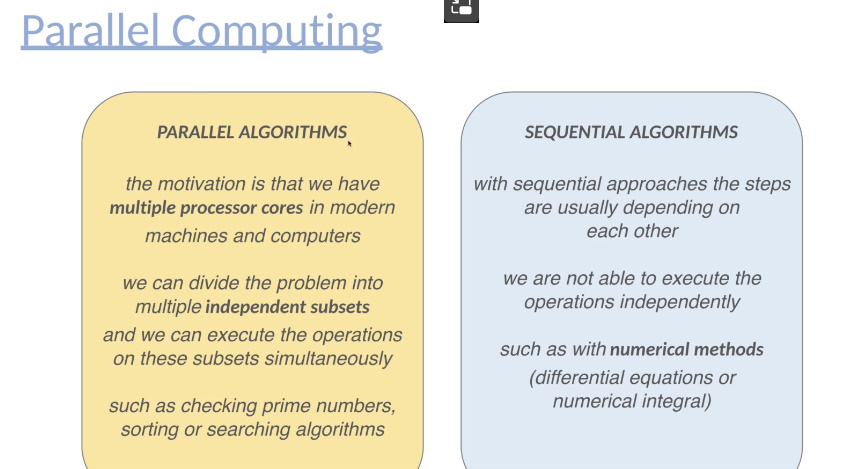
**Parallel Algorithms(Parallel Computing)**

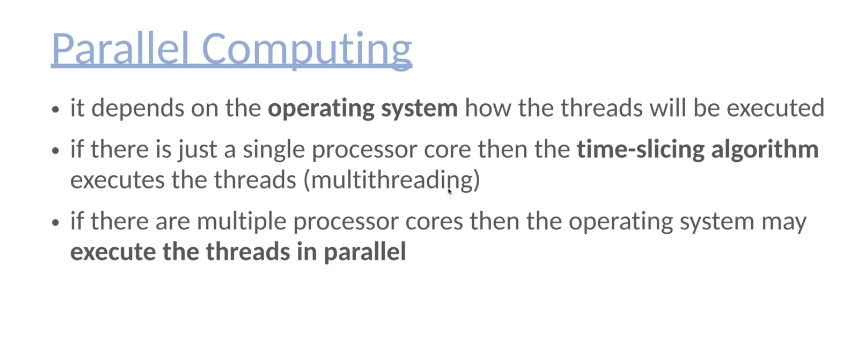
* Cand folosim abordarea **secventiala**, care e standarta, limbajul de programare executa operatiile linie cu linie, unul dupa altul
* Abordarea **multithreading** se bazeaza pe folosirea mai multor threduri, dar acestea sunt executate de time-slicing algorithm
* Multithreading e folosit pentru a executa taskuri independente fara a bloca alte taskuri
* Java colaboreaza cu OS pentru a se asigura ca Threadurile sale vor fi executate paralel, dar nu in toate situatiile asa ceva e posibil.
* Cu **parallel alghoritms**, executam diferite taskuri pe diferite cores in mod simultan. Deci, diferite taskuri vor fi executate chiar in acelasi timp!
* **Parallelization e folosit pentru a creste viteza unei aplicatii**
* **Multithreading nu poate fi folosit pentru a creste viteza aplicatiei!!!Scopul la multithreading nu este de a creste viteza unei aplicatii. Multithreading pur si simplu inseamna rularea a mai multe threaduri, si atat, si asta poate fi si pe un microprocesor single thread. Parallelization neaparat implica mai multe cores.**
* Unele probleme sunt secventiale implicit si nu putem folosi parallelization



* vedem ca in multithreading, 2 threaduri nu pot fi rulate exact in acelasi timp pe un core, si timpul lor de executie e impartit de time slicing algorithm.
* ***Multithreading inseamna rularea a mai multe threaduri si pe un core, dar parallelization implica mai multe cores***
* Dar, in cazul la parallelization, trebuie neaparat implicate mai multe cores. Fiecare thread va fi rulat de un core separat si ele chiar vor fi executate simultan.



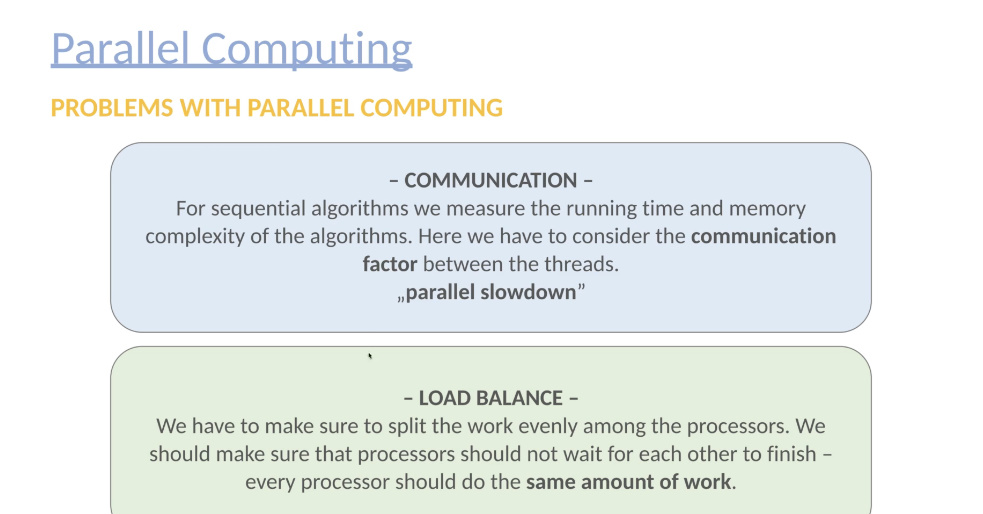
In algoritmii secventiali, fiecare pas din rezolvarea unei probleme depine de celalalte, dar nu si in parallel algorithms. Nu putem rula operatiile in mod independent.

* **Sequential Algorithm** sunt pentru problemele care necesita sa fie rezolvate pas cu pas, si un pas depinde de altul.Sa zicem ca avem o aplicatie care vrea sa afiseze un grafic pe baza unor date. Intai va trebuie ca un thread sa instaleze datele de pe net, si apoi pe baza acestor date va arata un grafic. Aici Parallel Algorithms nu ne ajuta, caci nu vrem sa se arate graficul cand datele inca sunt in curs de instalare
* **Parallel Algorithms** sunt pentru problemele in care rezolvarea lor nu are pasi ce depind unii de altii, ci sunt independenti, si pot fi rulati separat exact in acelasi timp.
* 

Deci, daca microprocesorul are mai multe cores, chiar de folosim multithreading, se poate intampla ca cateva threaduri sa fie executate chiar simultan, exact in acelasi timp, pe diferite cores. Deci, chiar de nu intentionam sa folosim parallelization, se poate intampla ca ea sa fie folosita fara sa stim.

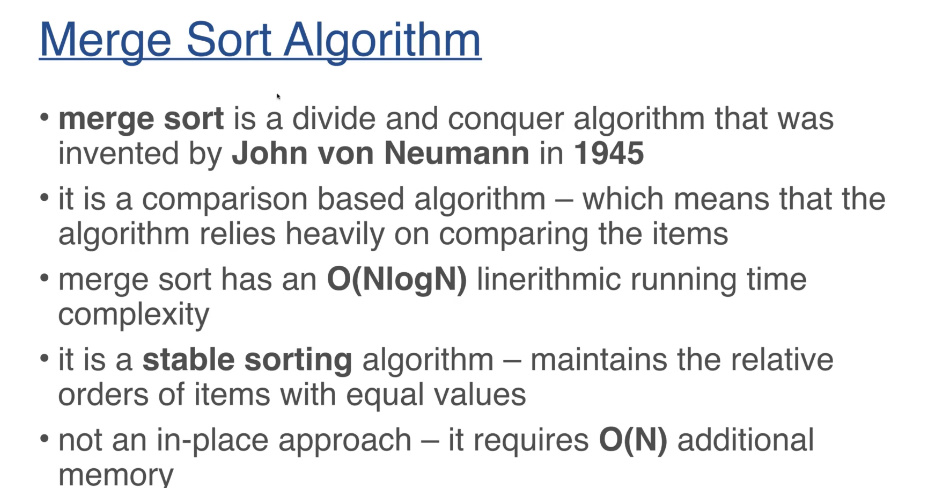
* **Nu putem scrie vreun code care sa asigure ca mai multe threaduri vor fi executate paralel pe diferite cores! Parallelization e ceva ce nu putem controla, dar se intampla pe microprocesoare cu multicores!!!Doar OS poate decide cum sa ruleze threadurile, si mai multe nimeni**

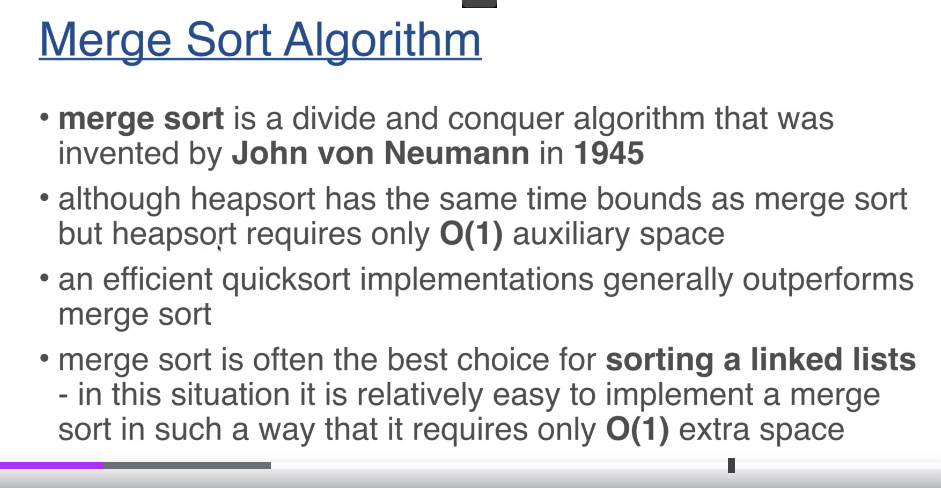
**Probleme cu parallelization**

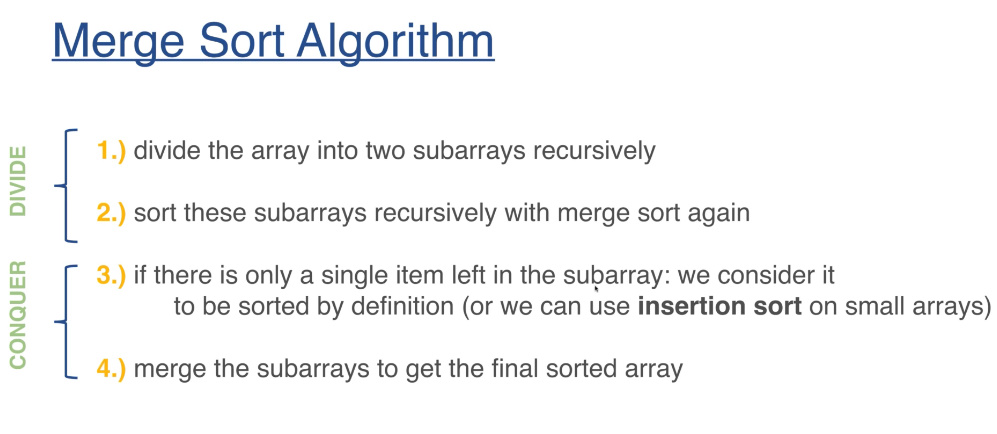
* 

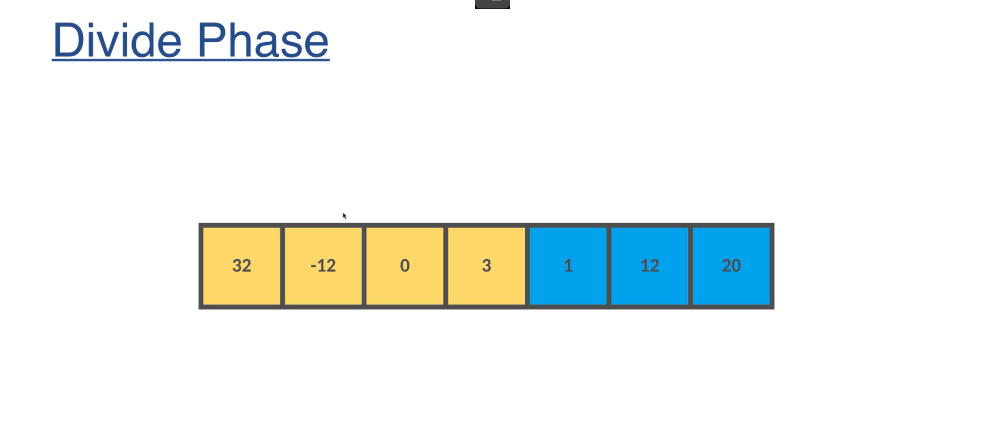
1. Threaduriel care ruleaza paralel trebuie sa comunice intre ele, si asta consuma multe resurse, de asta cand facem o analiza a unui algoritm, trebuei sa luam in calcul factorul de comunicare intre threaduri. Problemele simple ar putea lua mai mult timp sa fie rezolvate cu algoritmi paraleli decat cu algoritmi secventiali
2. Threadurile trebuie sa faca exact aceeasi cantitate de lucru, ca sa nu se primeasca ca un thread termina inaintea la altul. De ex, daca avem de a face niste operaratii grele asupra la un array cu 1 milion de elemente, e mai usor sa dam fiecarui thread cate o jumatate de array, dar se poate intampla ca prima jumatate sa fie din numere mici, si a 2 din numere foarte mari, si asa un thread va termina repede lucrul, dar altul greu.

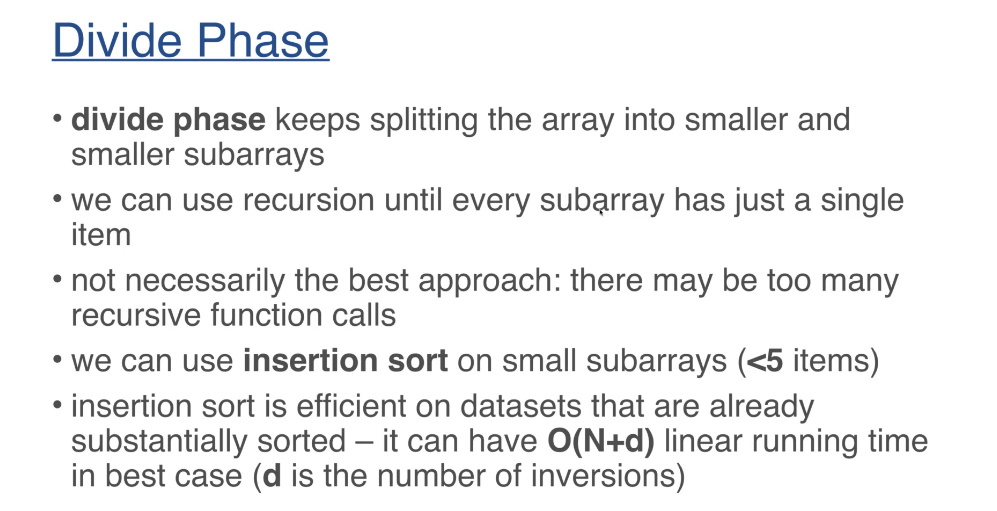
**Merge Sort**

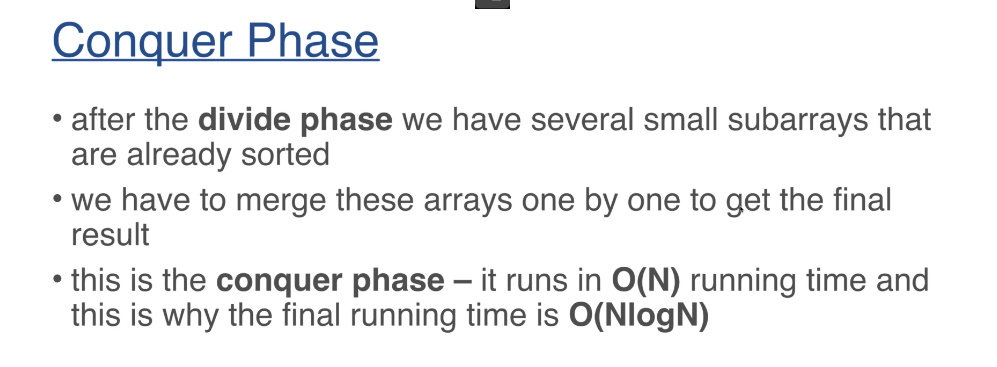


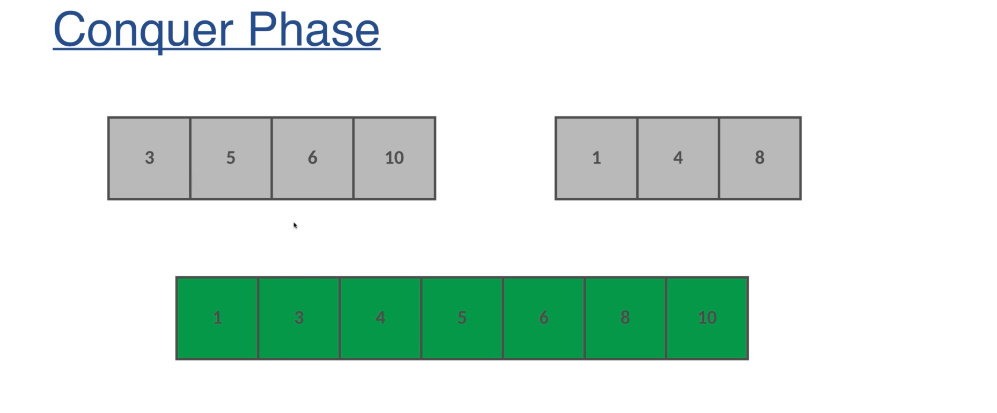












Deci, partea de **Divide** consta in divizarea arrayului pana ajungem la array de 1 element

Partea de **Conquer** e cand comparam elementele

Fie array 5 8 9 6 3 2 1

Size = 7

Mijloc = 7/2 = 3, deci index 3 are valoarea 6. Se imparte in 2

5 8 9 6 3 2 1

5 8 | 9 6 3 2 | 1

5 | 8 9 | 6 3 | 2 1

1. 5<8, deci ramane 5 8

9 > 6, deci va fi 6 9

3>2. deci va fi 2 3

5 8 9 6 3 2 1

5 8 | 6 9 2 3 | 1

5 | 8 6 | 9 2 | 3 1

1. 5 8 | 6 9

5 < 6, deci ramane 5. Comparam acum 8 cu restul, caci 5 nu mai are sens

8 > 6, trece 6 in locul lui

5 6 | 8 9

Avem 5 6 8 9

1. Facem asa si cu 2 jumatate din array:

2 3 | 1

2 < 1, deci trece 1

1 2 3

1. Avem acum de sortat

5 6 8 9 | 1 2 3

5 < 1, deci 1 trece

5 < 2, deci 2 trece

5 < 3, deci 3 trece

1 2 3 5 6 8 9

Luam primul element din primul array si il comparam cu primul din al 2. Daca elementul din array1 e mai mic ca primul din array2, atunci primul element din array1 sta unde trebuie, si luam al 2 element din array1, si iar il comparam cu primul din array2. Daca elementul 1 din array1 e mai mic, el trece in locul elementului 2 din array1, dar continuam sa comparam elementul inlocuit cu urmatoarele elemente din array2 ca sa vedem care vor trebui puse dupa cel deja adaugat in array1.

**Mai sus, daca numarul de elemente e par, middle intra in array, dar poate fi si ca middle sa intre in array2, nu conteaza!**

**dezavantaje la Merge Sort**

* Recursivitatea e lenta, si asta face algoritmul nu prea eficient, chiar daca are running time O(nlogn)
* Necesita de a crea multe variabile noi si inca un array

**Parallel Merge Sort**

* Merge sort poate fi rezolvat si cu Parallelization
* **Cand avem un procesor cu multicores si folosim multithreading, deja folosim Paralellization**
* Numarul de cores va reprezenta numarul de subarrays ce le sortam in mod paralel. Deci, nu vom crea atatea threaduri cate subarrays exista, caci comunicarea intre threaduri e scumpa si nici nu avem sute de cores
* Daca avem 2 cores, impartim array in 2 subarrays si vom avea 2 threaduri ce vor rula paralel, daca 3 cores, atunci cream 3 threaduri si impartim array in 3 parti
* Dar, daca avem de ex 3 threaduri, fiecare fiind responsabil de a treia parte din array, sau cel putin aproximativ a 3 parte, ele vor trebui sa ruleze toate in aceeasi perioada de timp, sau vor trebui sa se astepte toate, caci nu vom putea face merge final la toate 3 arrays sortate daca ele nu sunt sortate deja toate. Asa dar, prima parte din array poate fi sortata in 0,1ms , dar a 2 in 0,11 ms si a 3 in 0,12ms, deci nu putem crea tot array sortat daca numai o parte e sortata
* Acum, fiecare subarray din fiecare thread va fi sortat in mod secvential, nu paralel, ca sa nu cream threaduri pentru fiecare subarray creat in subbarays de baza

**RunTime**

* Fiecare aplicatie Java are un obiect de tip RunTime creat.
* El ne ofera posibilitatea de a afla informatii despre cores,
* **getRuntime**() – returneaza obiectul runtime
* *getRuntime*().availableProcessors()

vedem cate cores exista

* .**freeMemory**()
* .**maxMemory**

**Parallelization or Multithreading**

* Daca problema are putine date, parallelization poate face algoritmul mai lent, deoarece comunicarea intre threaduri si crearea lor ar putea lua mai mult timp ca insusi executarea algoritmului. Deci, parallelization e mai mult pentru problemele cu multe date si care evident, nu necesita strict un algoritm secvential
* Multithreading e mai bun pentru problemele care au putine date si sunt secventiale.