

Тема 9: Паралелни алгоритми. Класификация на  
многопроцесорните структури. Паралелизъм в  
алгоритмите. Примери за паралелни алгоритми  
- минимално скелетно дърво, сортиране и др.  
Генетични алгоритми

---

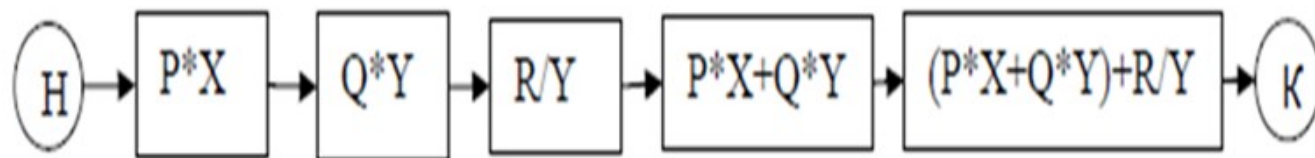
## Паралелни алгоритми

- При паралелните алгоритми в даден момент от време се изпълняват повече от една инструкции - това е един от подходите за повишаване на бързодействието на компютърните системи.
- Другият подход е повишаването на бързодействието на елементната база.
- Паралелните алгоритми се изпълняват предимно от мултипроцесорни системи. Те имат един съществен недостатък - не е възможно всички процесори да работят едновременно непрекъснато при изпълнение на даден алгоритъм (или на няколко алгоритъма) - една част от тях не работят, т.е. не се използват ефективно.

Нека е даден израз  $U = P * X + Q * Y + R / Y$ .

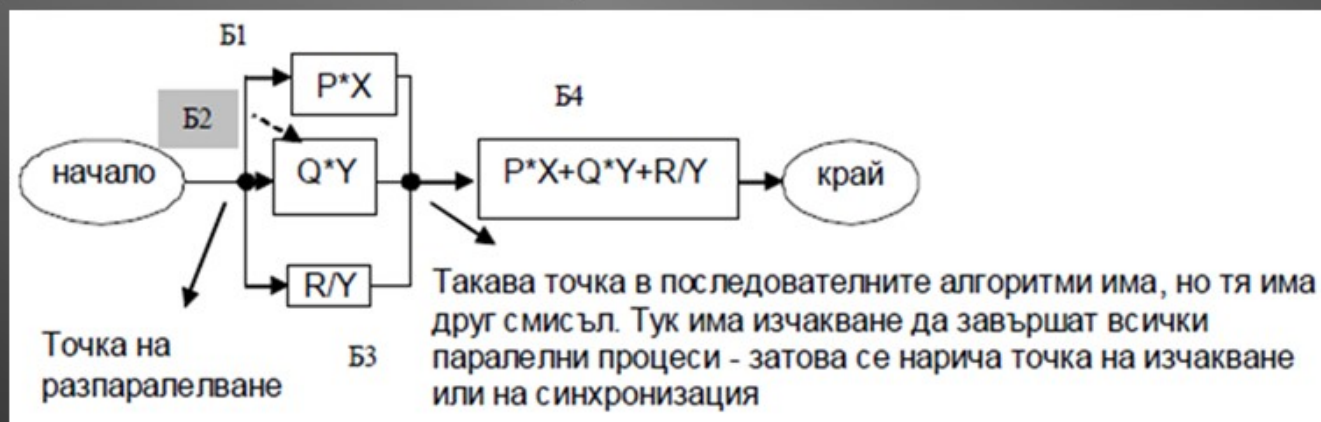
Изчисляването на този израз може да стане:

- а) с последователен алгоритъм:



- б) с паралелен алгоритъм:

- В този алгоритъм едновременно (*паралелно*) се изпълняват блокове Б1, Б2, Б3
- Когато завърши изпълнението на всеки от тях, тогава започва изпълнението на блок Б4.
- Различието спрямо последователните алгоритми са точките, в които започва (*разпаралелване*) и завършва (*синхронизация*) паралелното изпълнение на операции.





## **Класификация на мултипроцесорните системи по Флин:**

- извършва се по броя на потоците от инструкции, които се изпълняват едновременно и броя на потоците от данни, които се обработват едновременно.
- Използват се следните означения:

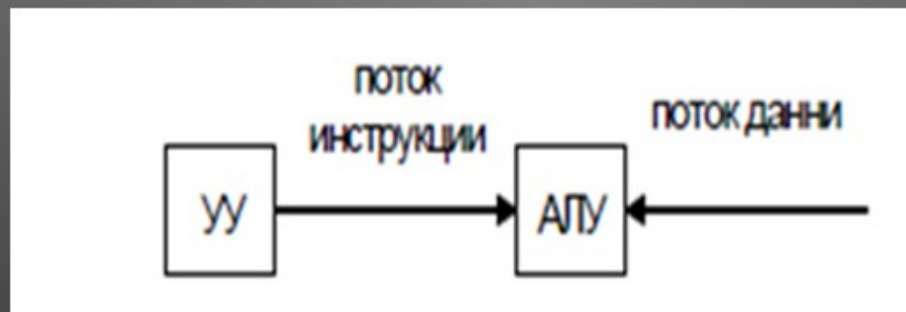
УУ - управляващо устройство,

АЛУ - аритметично-логическо устройство,

S - single, M - multiple, I - instruction, D - data.

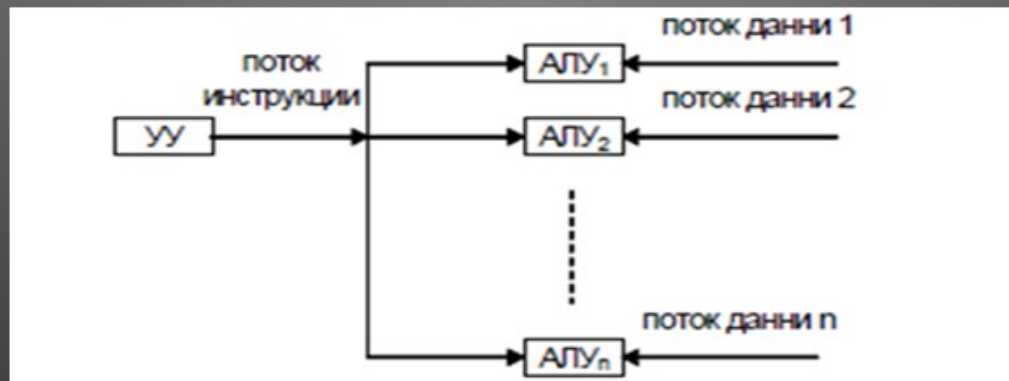
# 1. SISD (SINGLE INSTRUCTION - SINGLE DATA)

Управляващото устройство генерира един поток от инструкции и се обработва един поток от данни.  
Това е **класическият фон Нойманов модел** на еднопроцесорна система.



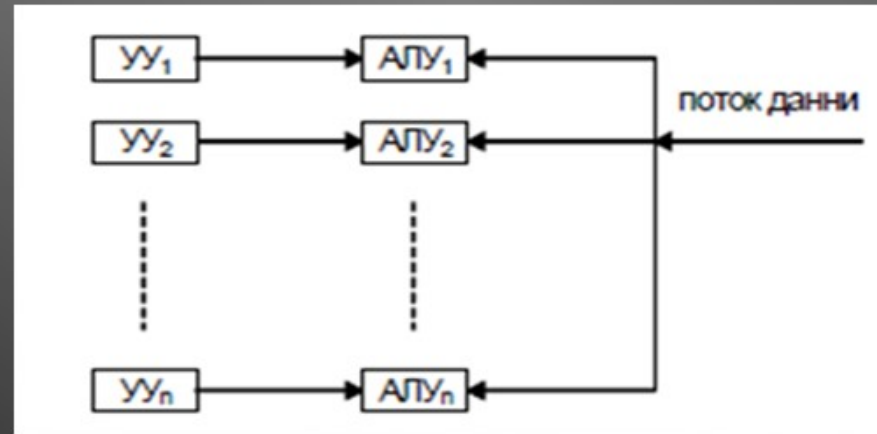
## 2. SIMD (SINGLE INSTRUCTION - MULTIPLE DATA):

Управляващото устройство генерира 1 поток от инструкции, който се изпълнява от всички АЛУ-та за различните потоци от входни данни.



### 3. MISD (MULTIPLE INSTRUCTIONS - SINGLE DATA):

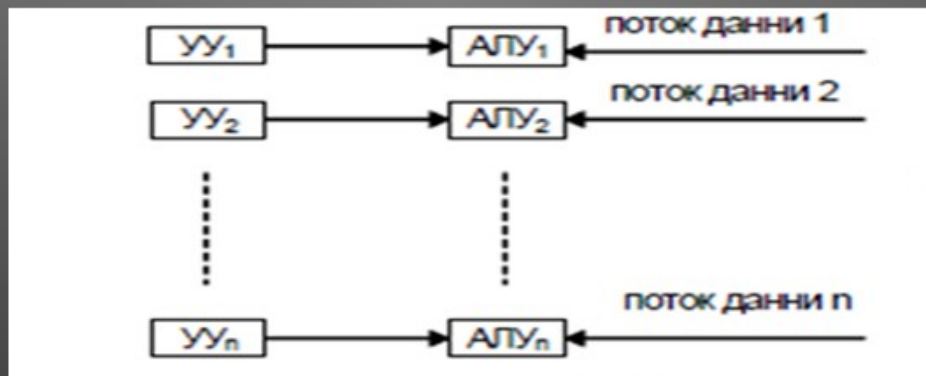
Множество инструкции се изпълняват върху един поток входни данни.





## 4. MIMD (MULTIPLE INSTRUCTIONS - MULTIPLE DATA):

Множество потоци от инструкции се изпълняват върху много потоци входни данни



## Примери на паралелни алгоритми:

**Пример 1:** Определяне на сумата на елементите на едномерен масив:

- $S = A[1] + A[2] + A[3] + A[4] + \dots + A[n-1] + A[n];$

Последователното изчисление на израза е чрез

- $S := A[1]; \text{ for } i := 2 \text{ to } n \text{ do } S := S + A[i];$

Паралелното изчисление се осъществява като всяка двойка събираеми се подава на отделен процесор.

- Нека  $n=16$ .

- Тогава, в началото **8 процесора** ще определят сумите  $S_{1,2} = A[1] + A[2]$ ,  $S_{3,4} = A[3] + A[4]$  и т.н.,

- след това **4 процесора** ще определят сумите  $S_{1,2,3,4} = S_{1,2} + S_{3,4}$ ,  $S_{5,6,7,8} = S_{5,6} + S_{7,8}$  и т.н.,

- след което **2 процесора** ще определят сумите  $S_{1,2,3,4,5,6,7,8} = S_{1,2,3,4} + S_{5,6,7,8}$ ,  $S_{9,10,11,12,13,14,15,16} = S_{9,10,11,12} + S_{13,14,15,16}$

- и накрая **1 процесор** ще определя крайната сума  $S$  на последните 2 частични суми.

Основният проблем при паралелните алгоритми са циклите – разпаралелването на циклите се усложнява, когато отделните изпълнения на тялото на даден цикъл са зависими, т.е. имат общи променливи.

***Пример 2: Паралелен алгоритъм за определяне на минималното скелетно дърво на граф***

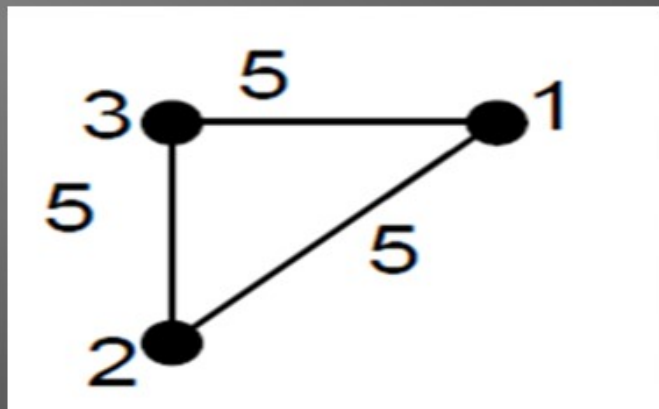
- Минимално скелетно дърво на тегловен граф наричаме дърво с минимално общо тегло на клоните, включващо всички възли на графа.
- В началото на всеки възел съответства процесор (за граф с неголям брой възли – за граф с голям брой възли на 1 процесор съответстват няколко възела).
- За да се избегне избор на клони, водещ до образуване на цикли (а това е недопустимо за дърво) се прилага следното правило:

***Ако има няколко клона с еднакво минимално тегло, свързани с даден възел, се взима този към възел с по-малък номер.***

Така се получава еднозначност на избора.

За посочения граф:

- процесорът на възел 1 определя клон 1-2,
- процесорът на възел 2 определя същия клон,
- процесорът на възел 3 определя клон 1-3.





## Стъпки на алгоритъма:

**C1 :** На всеки възел е присвоен процесор и всеки процесор определя едновременно клона с минимално тегло, излизащ от съответния му възел. При няколко клона с еднакво тегло се взема този, водещ към възел с по-малък номер. На променливата **K** се присвоява броя на избраните клони.

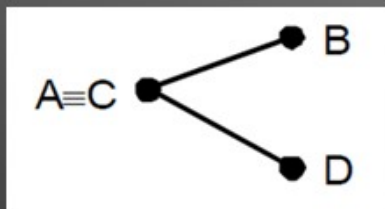
**C2 :**

- **while**  $k < n-1$  **do**
- **begin**
  - Сега на всяко поддърво е присвоен процесор. Едновременно за всяко поддърво **T** на графа, получено от избраните до момента клони, се избира от съответния му
  - процесор клон с минимално тегло. Този клон се избира измежду всички клони,
  - свързващи възел от **T** с възел от друго поддърво. В случай на няколко клона с минимално тегло се избира клон, който е свързан към възел с най-малък номер, а когато няколко клона са свързани с този възел, се избира клон, на който другият краен
  - възел е с най-малък номер. На **K** се присвоява броят на вече избраните възли.
- **end;**

Нека да поясним по-подробно случая с 2 клона (A,B) и (C,D) с минимално тегло: клоните са записани така, че  $A < B$ ,  $C < D$

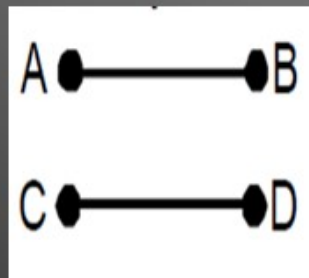
### I. Случай

- Ако  $B < D$ , се избира клоната (A,B); ако  $B > D$ , се избира (C,D)



### II. Случай

- $A \neq C$
- Ако  $A < C$  се избира (A,B)
- Ако  $C < A$ , се избира (C,D)



## Пример (за графа отляво на фигурата):

### C1 :

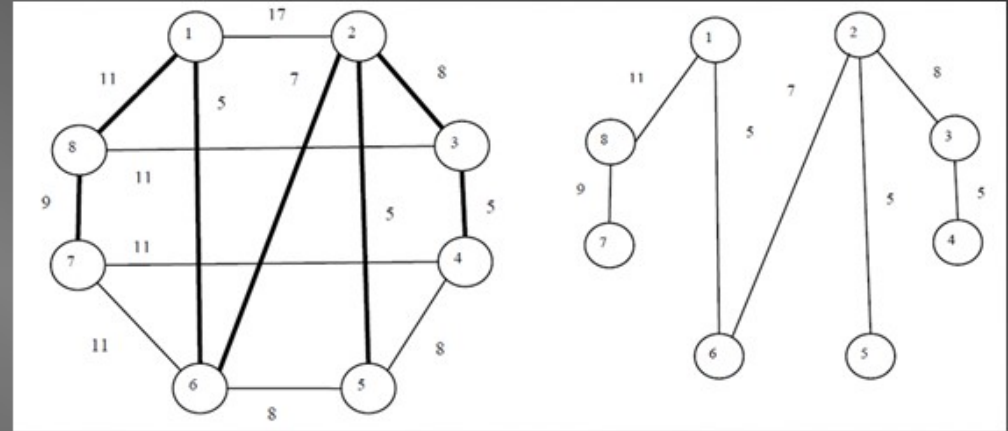
- От всеки възел се избира клонът с минимално тегло:
- (1,8) с тегло 5,
- (2,5) с тегло 5,
- (3,4) с тегло 5 и
- (7,8) с тегло 9;
- $K=4$ ;

### C2 :

- От поддърво (1,6) се избира клон (2,6) с тегло 7;
- От поддърво (2,5) се избира клон (2,6) с тегло 7;
- От поддърво (3,4) се избира клон (2,3) с тегло 8;
- От поддърво (7,8) се избира клон (1,8) с тегло 11, тъй като от 4-те клона с минимално тегло ((1,8) (3,8) (4,7) (6,7)) той е към възел с най-малък номер;
- Теглото на минималното скелетно дърво (почернените клони, а също и фигурата отдясно) е  $5+5+5+9+7+8+11=50$ .

**Сложността на последователния алгоритъм (Крускал или Прим) е  $O(n^2)$ , а на паралелния  $O(n \log n)$ .**

**Други интересни примери на паралелни алгоритми могат да бъдат за сортиране, за бързото преобразуване на Фурие.**



# Генетични Алгоритми

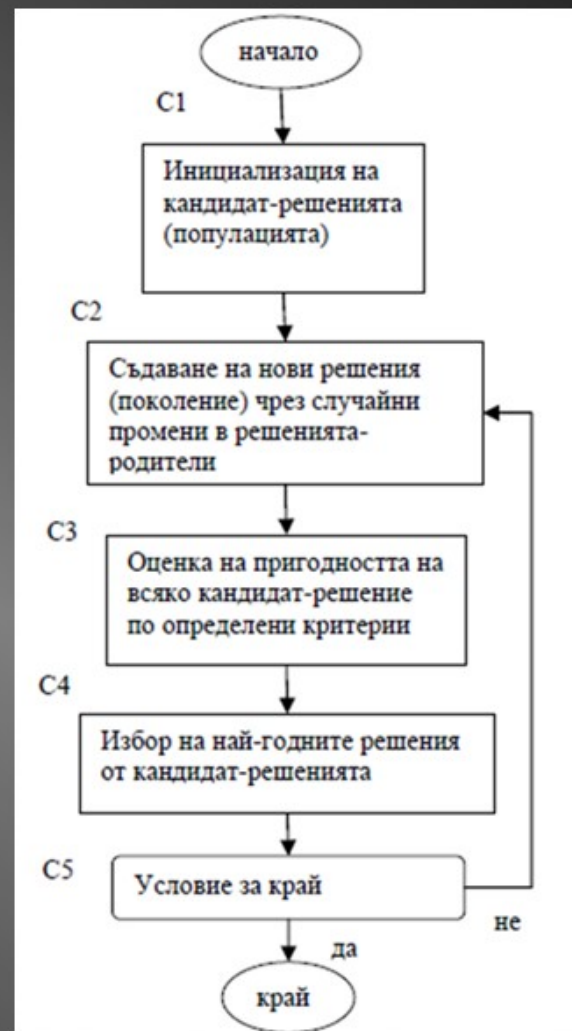
## 1. Основа на генетичните алгоритми:

- Идеята на тези алгоритми е заимствана от биологията – това е принципа на естественият подбор при еволюцията на организмите.
- Съгласно този принцип всеки организъм е резултат на верига от исторически събития и влияние на околната среда (наследственост, изменчивост и естествен подбор).
- Под въздействието на променящите се условия организмите постоянно се изменят.
- Малките изменения се наследяват и ако са полезни за организмите се задържат от естествения подбор. Така те постепенно се натрупват и засилват.



## 2 Примерна блок-схема на генетичен алгоритъм:

- В началото в генетичния алгоритъм се генерират случайни решения (родители) – от тях се генерират нови решения (ново поколение), които се оценяват по определени критерии и се вземат по-добрите, които в следващия цикъл играят ролята на родители.
- Това продължава толкова на брой цикли (поколения), колкото е необходимо за намиране на решение близко до оптималното в достатъчна степен.
- Типичният генетичен алгоритъм има вида, показан на блоковата схема.



## Пример на генетичен алгоритъм:

**Стъпките на генетичния алгоритъм за решаване на TSM задача могат да бъдат следните:**

**C1)** генериране на  $n$  случайни цикъла (пермутации) – решения-родители.

**C2)** генериране на нови решения. Това може да стане по следните 2 начина:

**А)** от всяка пермутация-родител се получава нова пермутация чрез случаен избор на участък от пермутацията, който след това се инвертира. Например, от пермутация **5 3 6 1 8 2 4 7** (при  $n=8$ ) при случаен избор на участък **6 1 8 2** се получава нова пермутация **5 3 2 8 1 6 4 7**. Този начин съответства на еднополовото размножаване при организмите.

**Б)** от 2 пермутации-родители се получава нова пермутация чрез случаен избор на позиция, по която всяка от пермутациите се разделя на 2 части. Новата пермутация се получава, например чрез сливане на първата част от първата пермутация-родител и втората част на втората пермутация-родител. Ако, при това, се получи повторение на номер на възел в пермутацията то трябва да се отстрани, примерно чрез замяна с някои от липсващите номера.

**Например**, от пермутации **5 7 2 4 1 6 8 3** и **7 2 3 5 8 1 6 4**, и случаен избор на позиция 3 се получава следното разделяне на части на пермутациите

**5 7 2 4 1 6 8 3, 7 2 3 5 8 1 6 4**, от които се получава **5 7 2 5 8 1 6 4**, където има повторение на 5.

То може да се отстрани чрез замяна с липсващия номер 3 и окончателно се получава **5 7 2 3 8 1 6 4**.

Този начин съответства на двуполовото размножаване при организмите.

**C3)** Оценка на решенията: за всяка пермутация (цикъл) се определя сумата от теглата на клоните ѝ.

**C4)** Избор: от всички пермутации (родители и нови) се избира част (примерно половината). Това са тези пермутации, които има по-малко тегло от другите, които не са избрани.

**C5)** Условие за край – след изпълнението на зададен брой цикли алгоритъмът прекратява изпълнението си.

# Многонишково програмиране (Scalable Multithreaded Programming) с „Thread Pools“ (паралелна сортировка )

Съществуват много начини за разпределяне задачите по процесорни ядра или в многопроцесорна среда.

Ще преобразуваме еднонишково приложение към такова, ползващо всички налични изчислителни ресурси.

Ще въведем основни понятия от:

- OpenMP технологията и
- --thread pools.

Ползвайки **Visual Studio utilities**, ще демонстрираме измерване на подобренията в производителността.



В началото, ще раздробим общата задача на по-малки, подходящи за нишково разпределяне подзадачи, които бихме могли да подадем към отделните ядра.

Спазваме няколко препоръки при това преобразуване:

- Задачата да подлежи на разпаралеляване.
- Подзадачите да са независими помежду си.
- Да могат да се изпълняват в произволен ред.
- Да поддържат собствени копия на данните си.



## 3.1 Multithreading с OpenMP

OpenMP е една от най-простите технологии за въвеждане на паралелизъм и се поддържа от Visual Studio C++ компилатора след версия 2005. Въвеждането на паралелизма става чрез „OpenMP - pragmas“ директивите в кода. Пример:

```
include <omp.h>                                // You need this or it won't work
#include <stdio.h>

int main (int argc, char *argv[]) {
    #pragma omp parallel
    printf("Hello World from thread %d on processor %d\n",
           ::GetCurrentThreadId(), ::GetCurrentProcessorNumber() );
    return
}
```

директивата ‘**pragma**’ паралелизира следващия я блок код— в случая това е само printf()— и го пуска едновременно по всички изчислителни ресурса.

Броят им варира - според инсталираните процесори или ядра.

---

За да разрешите паралелизацията с OpenMP ( компилаторът да не игнорира „pragma“ директивите), следва да разрешите OpenMP. За целта: първо

**опция КЪМ КОМПИЛАТОРА /openmp**

(Properties | C/C++ | Language | Open MP Support). Второ: **include the omp.h**



Следва пример за разпаралеляване на матрични изчисления (повдигане на степен) и обработка върху всички налични ядра:

```
#pragma omp parallel for  
for (int i = 0; i < 50000; i++)  
    array[i] = i * i;
```

OpenMP притежава и конструктори за контрол на:

- броя създавани нишки;
- Управление последователността на отделните запаралелени блокове;
- Създаване на thread-local data,
  - точки на синхронизация,
  - критични секции и др.

OpenMP е лесен начин за въвеждане на паралелизъм в съществуващ вече код.

OpenMP е проста технология. В много случаи е необходим по-голям контрол над изчислителния процес.

Например, разпределяне нишките по конкретни ядра, динамичен паралелизъм и т.н..

Идват други технологични решения –например, **thread-pools**.

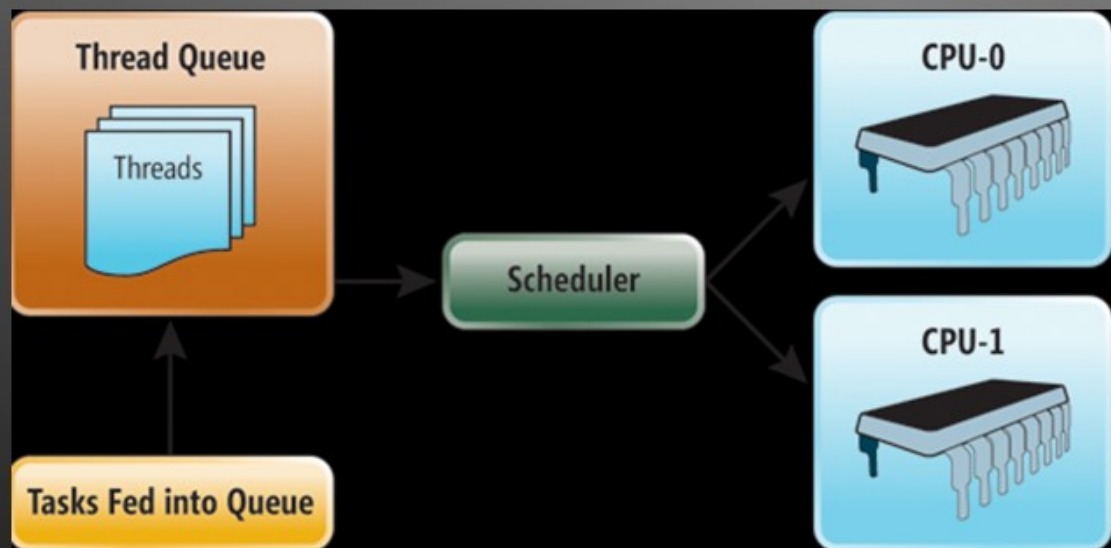


# Thread Pool

Нишките се менажират от OS. Те заемат ресурс и създават свое обкръжение. Следователно, честото им създаване и унищожаване е скъпа операция. По-добре е вече създадените да отлежават за повторно използване, при необходимост.

Това място за изчакване се нарича **thread pool** и Windows поема поддръжката му .

Ползването на thread pool сваля от програмиста отговорността за често създаване на нишки , унищожаването и управлението им.



Удобно и полезно е да се разпаралели задача в много ядра.



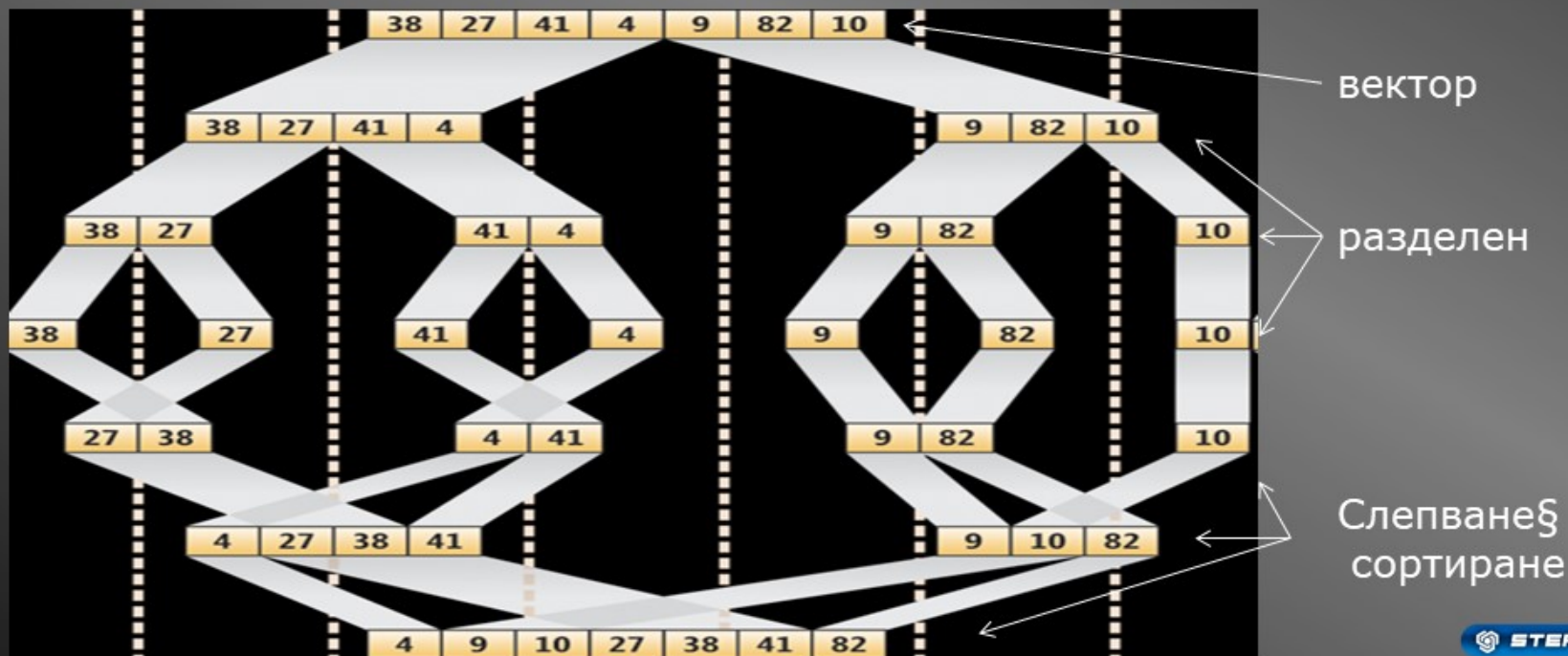
### 3.3 пример с многонишково сортиране

Сортировката не е най-силния алгоритъм за паралелизация. Тук има пречка – как да разпаралеляваме, така че да подсигуририм независимост на отделните блокове?

Наивен подход е да заключим достъпа до данните за времето на обработка например чрез mutex, semaphore или критична секция.

По-добро решение е на всяко ядро да се подаде подсекция на масива данни за паралелна сортировка. Този **divide-and-conquer подход** е като че ли по-подходящ.

Алгоритмите **merge sort** и **quick sort** работят добре с тази стратегия и тя е подходящо решение за натоварване на многоядрена изчислителна среда.





Разделените подписъци са независими и могат да се подадат към CPU ядрата за паралелна обработка без заключвания.

Съществуват много алгоритми за сортировка, податливи на Паралелизация:

- Quick sort,
- selection sort,
- merge sort,
- radix sort

Всички те разбиват данните и ги обработват независимо.

Примерна реализация на сортировка е показана (quick sort на C# ).

Програмата инициализира голям масив с поредица случайни числа и после ги сортира с помощта на quick sort процедура, като отчита и времето за това.



1

```

namespace ParallelSort {
    class Program {
        // For small arrays, use Insertion Sort
        private static void InsertionSort(
            int[] list, int left, int right)
        {
            for (int i = left; i < right; i++)
            {
                int temp = list[i];
                int j = i;
                while ((j > 0) && (list[j - 1] > temp))
                {
                    list[j] = list[j - 1];
                    j = j - 1;
                }
                list[j] = temp;
            }
        }

        private static int Partition( int[] array, int i, int j)
        {
            int pivot = array[i];
            while (i < j) {
                while (array[j] >= pivot && i < j)
                    j--;
                if (i < j) { array[i++] = array[j]; }
                while (array[i] <= pivot && i < j)
                    i++;
                if (i < j) { array[j--] = array[i]; }
            }

            array[i] = pivot;
            return i;
        }
    }
}

```

2

```

static void QuickSort( int[] array, int left, int right)
{
    // Single or 0 elements are already sorted
    if (left >= right)        return;

    // For small arrays, use a faster serial routine
    if ( right-left <= 32) { InsertionSort(array, left, right);
                            return;    }

    // Select a pivot, then quicksort each sub-array
    int pivot = Partition(array, left, right);
    QuickSort(array, left, pivot - 1);
    QuickSort(array, pivot + 1, right);
}

static void Main(string[] args)
{
    const int ArraySize = 50000000;
    for (int iters = 0; iters < 1; iters++)
    {
        int[] array;
        Stopwatch stopwatch;
        array = new int[ArraySize];
        Random random1 = new Random(5);
        for (int i = 0; i < array.Length; ++i)
            { array[i] = random1.Next(); }
        stopwatch = Stopwatch.StartNew();
        QuickSort(array, 0, array.Length - 1);
        stopwatch.Stop();

        Console.WriteLine("Serialt: {0} ms",
            stopwatch.ElapsedMilliseconds);
        .....
    }
}

```

За да паралелизирате приложението, променете следните оператори:

```
QuickSort( array, lo, pivot - 1);  
QuickSort( array, pivot + 1, hi);
```

Към паралелната реализация:

```
Parallel.Invoke(  
    delegate { QuickSort(array, left, pivot - 1); },  
    delegate { QuickSort(array, pivot + 1, right); }  
);
```

Интерфейсът **Parallel.Invoke** от **Systems.Threading.Tasks** namespace е дефиниран в **.NET Task Parallel Library**. Той позволява дефиниране на асинхронно изпълнявана функция. В нашата реализация - всяка отделна сортировъчна функция ще се изпълни в отделна нишка.

Макар че е по-добре да запустете само 1 нишка, а втория подписък да се поеме за сортиране от текущата, реализация с 2 запуснати нишки за сортировка е по-симетрична и илюстрира колко лесно е преобразуването на серийна програма в паралелен еквивалент.

Разумен въпрос е: паралелизацията подобри ли производителността?

Visual Studio 2022 включва няколко tools за такава оценка :


**Visual Studio 2022** мери производителност в паралелна реализация по време на изпълнение.

Можете да наблюдавате и core utilization.

Нека тестовата програма стартира серийна сортировка, заспива секунда и тогава стартира паралелна версия на сортировъчния алгоритъм.

За 4-ядрен компютър , получваме следната графична представа:

В началото е видно натоварването при 1 ядрена реализация – 100% използваемост на единственото ядро. Впоследствие се вижда **2.25кратно ускорение**.

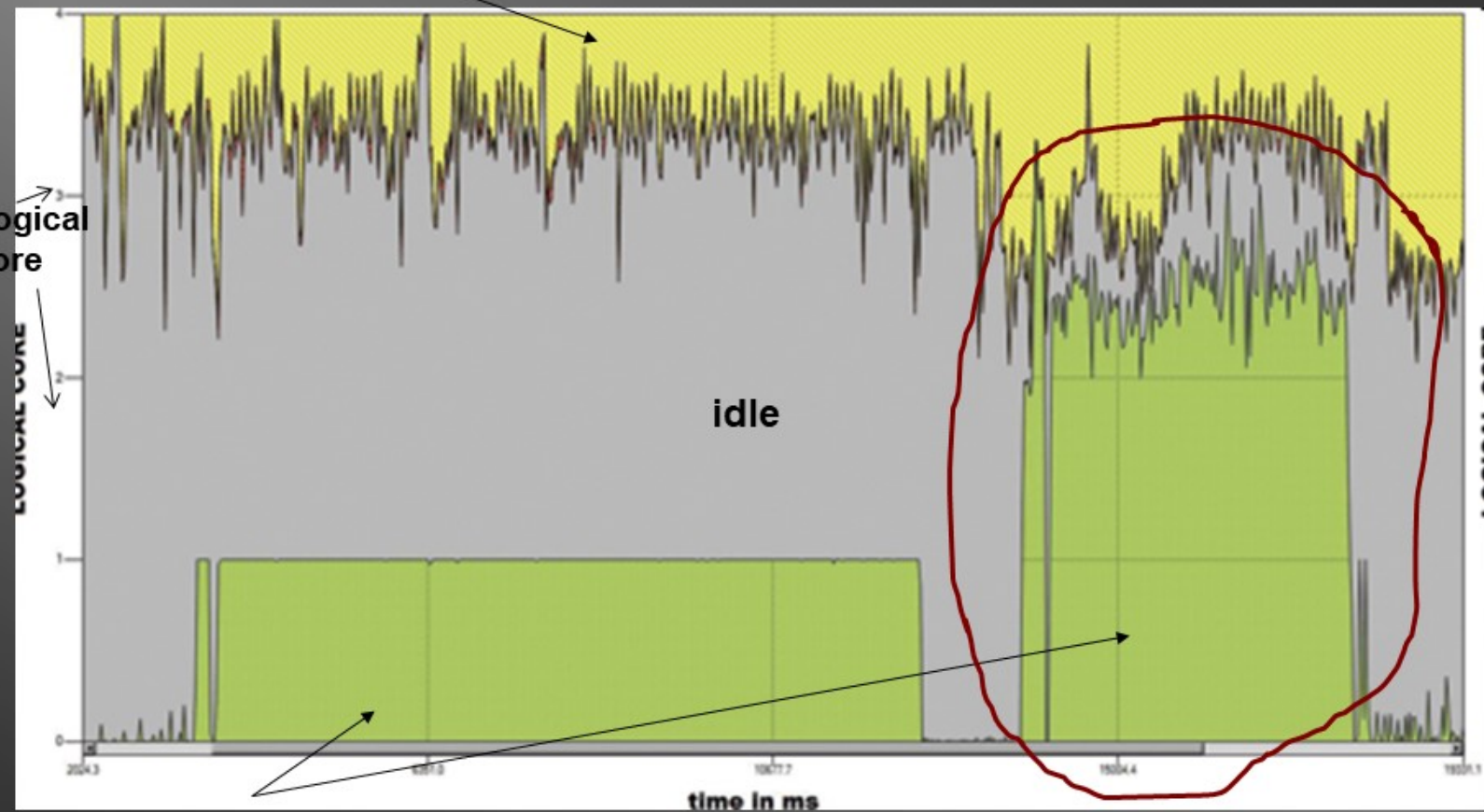


Паралелната реализация ускорява с около 45% в сравнение със серийния си еквивалент.



OS и други програми

Logical  
core



приложението

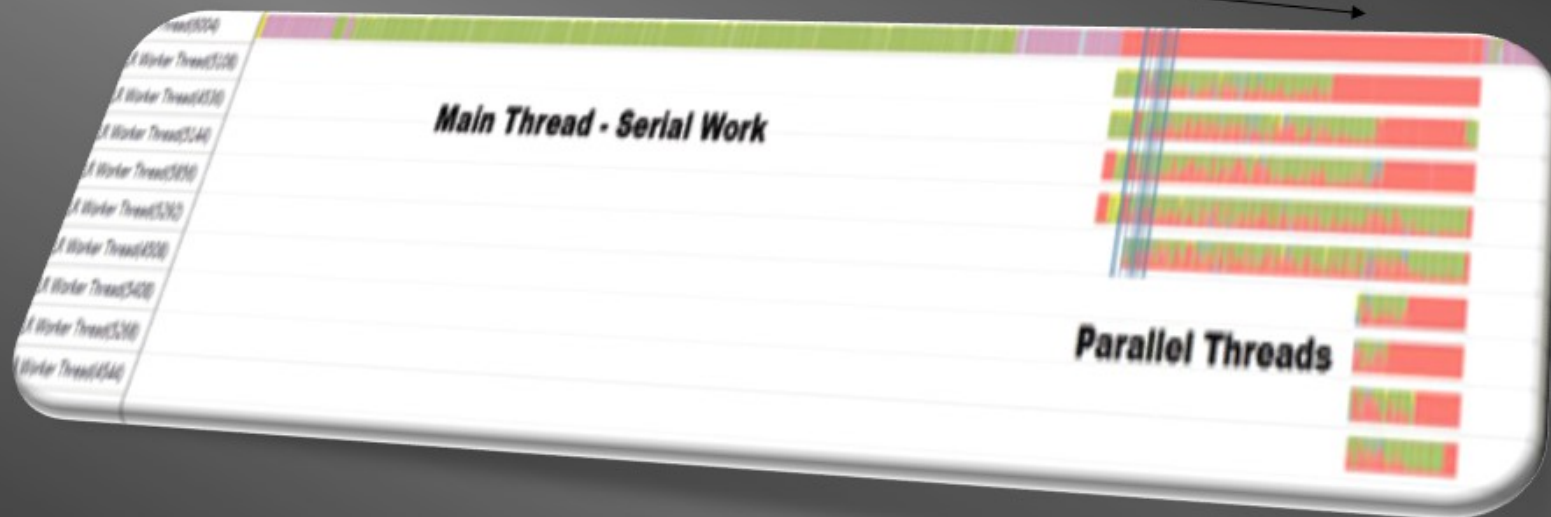


## Друга визуализация

Показваме как приложението ползва наличните нишки.

Една нишка работи почти цялото време.

Розовият цвят показва нишки, които са блокирани от други.



Вижда се , че макар натоварването на CPU ядра е подобро, има потенциал за още значителни оптимизации.

