



# *Паралелно Програмиране*

Теоретични аспекти на паралелните алгоритми.

Анализ на паралелни алгоритми.

Зависимости на данните, структурата и контрола.

# *PRAM Модел*

# PRAM модел

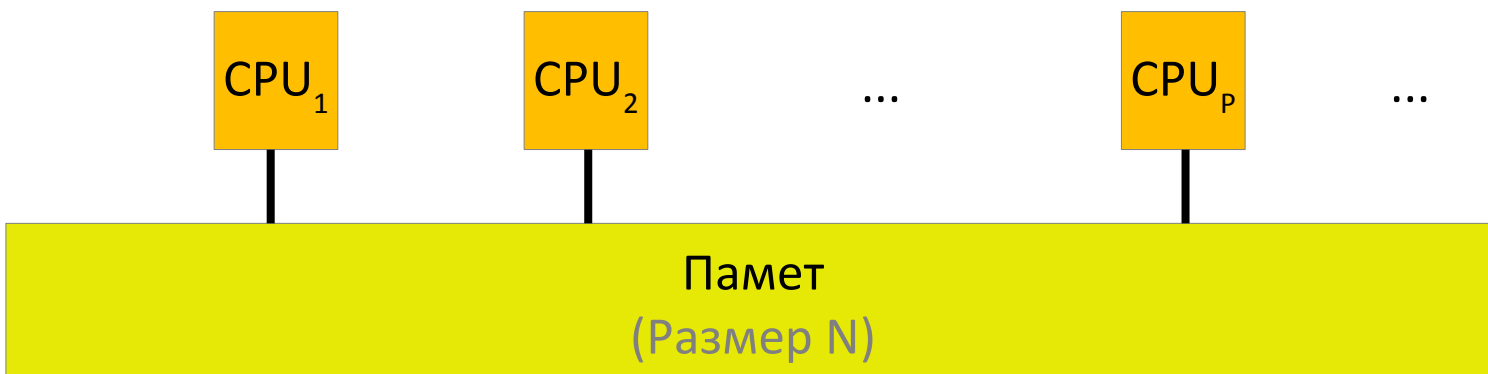
- ❖ Parallel Random-Access Machine (PRAM) е абстрактна машина със Споделена памет (Shared memory);
- ❖ Моделът пренебрегва практически проблеми като синхронизация и комуникация (по същия начин както RAM модела пренебрегва разликите във времената за достъп до кеша и основната памет);
- ❖ Моделът “осигурява” необходимия за решаване на проблема брой (всеки брой) процесори;
- ❖ Например, “цената” на алгоритъма се изчислява, като се използват два параметъра  $O(\text{време})$  и  $O(\text{време} \times \text{брой процесори})$ .

# *PRAM модел – допускания*

Този модел е опростен, защото са направени следните допускания:

- ❖ Няма ограничение на броя процесори в машината;
- ❖ Всяки адрес в паметта е равноправно достъпен за всеки процесор;
- ❖ Няма ограничение за обема на споделената памет в системата;
- ❖ Няма спорове за ресурси;
- ❖ Програмите написани за този тип машини са най-общо с SIMD инструкции.

# *PRAM*



# Конфликти при Четене/Запис

Конфликтите при едновременно четене/запис от един и същ адрес на споделената памет се разрешават по една от следните стратегии:

- ❖ Exclusive Read Exclusive Write (EREW) – всяка клетка от паметта може да бъде четена или записвана само от един процесор в даден момент;
- ❖ Concurrent Read Exclusive Write (CREW) – много процесори могат да четат от клетката, но само един може да пише в даден момент;
- ❖ Exclusive Read Concurrent Write (ERCW) – не се използва;
- ❖ Concurrent Read Concurrent Write (CRCW) – много процесори могат да четат и пишат едновременно;



# Конфликти при Четене/Запис

Четенията не водят до проблеми, докато конкурентните записи могат да създадат проблеми и се класифицират като:

- ❖ Общи (Common) – всички процесори записват една и съща стойност; (ако не е така е недопустимо/проблем)
- ❖ Произволни (Arbitrary) – само един произволен опит е успешен, а останалите се отхвърлят;
- ❖ Приоритетен (Priority) – ранга на процесора показва кой ще запише;
- ❖ Друг вид – например при различните видове редукции на масиви може да се използват операции като SUM, AND, MAX, ...

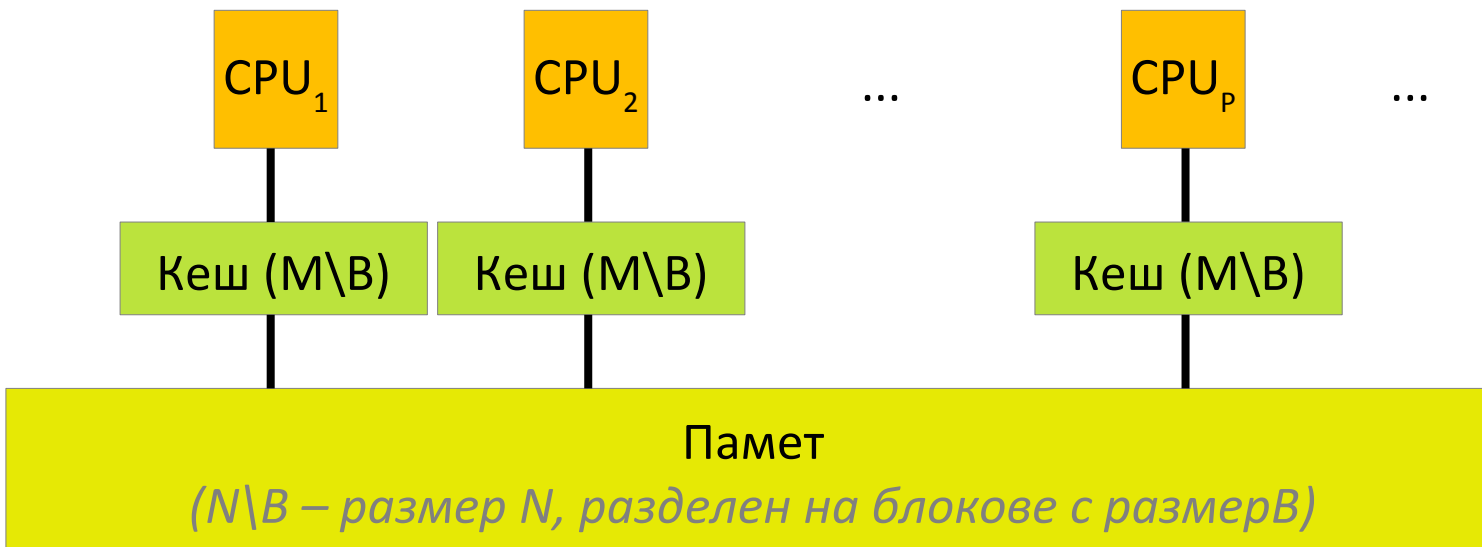
# *PEM Модел*



# *PEM модел*

- ❖ Parallel external memory (PEM) model е абстрактна машина с външна памет, базирана на кеш;
- ❖ Това е паралелно-изчислителна аналогия с модела на еднопроцесорна машина с външна памет (EM);
- ❖ По подобен начин е аналогията с кеша-базирана паралелна машина с произволен достъп (PRAM);
- ❖ Моделът PEM се състои от редица процесори, заедно със съответните им частни кеш и споделена основна памет.

# РЕМ модел

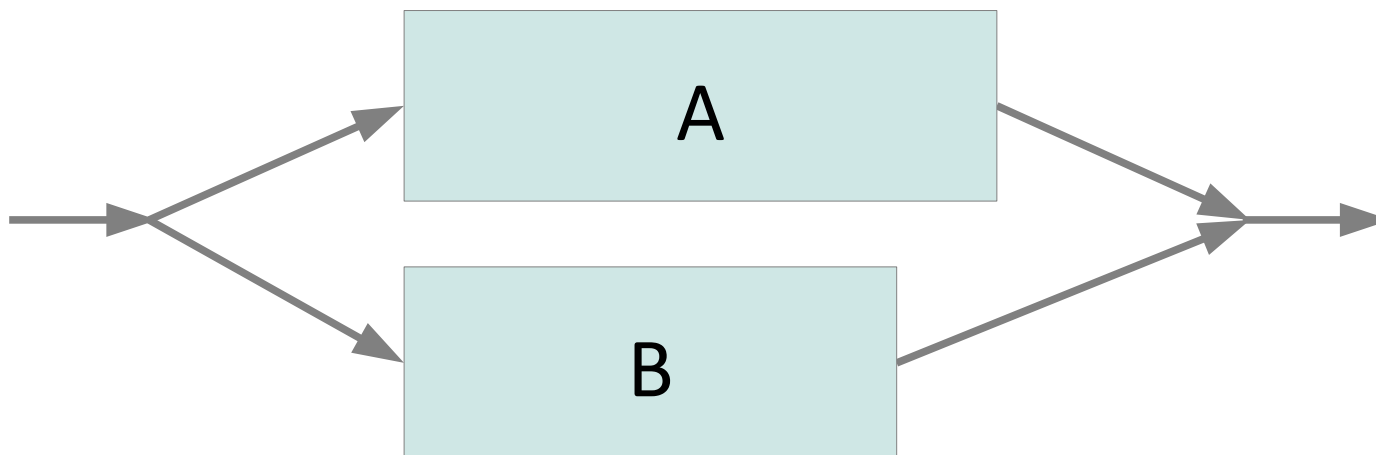
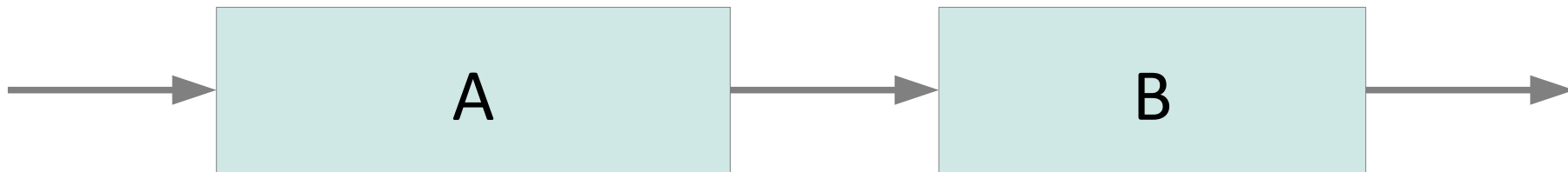


# *Разпаралеляване*

# Какво е разпаралеляване (паралелезация)?

- ❖ Паралелизация (parallelization) е процес на преобразуване на програма изпълняваща стъпките на алгоритмите си последователно в програма изпълняваща ги (там където е възможно) паралелно/едновременно;
- ❖ Паралелното изпълнение може да е с използването на SIMD инструкции (векторизация), много нишки, много процеси, много компютри;
- ❖ Обикновено се използва повече от едно ядро/процесор.

# Какво е разпаралеляване (паралелезация)?

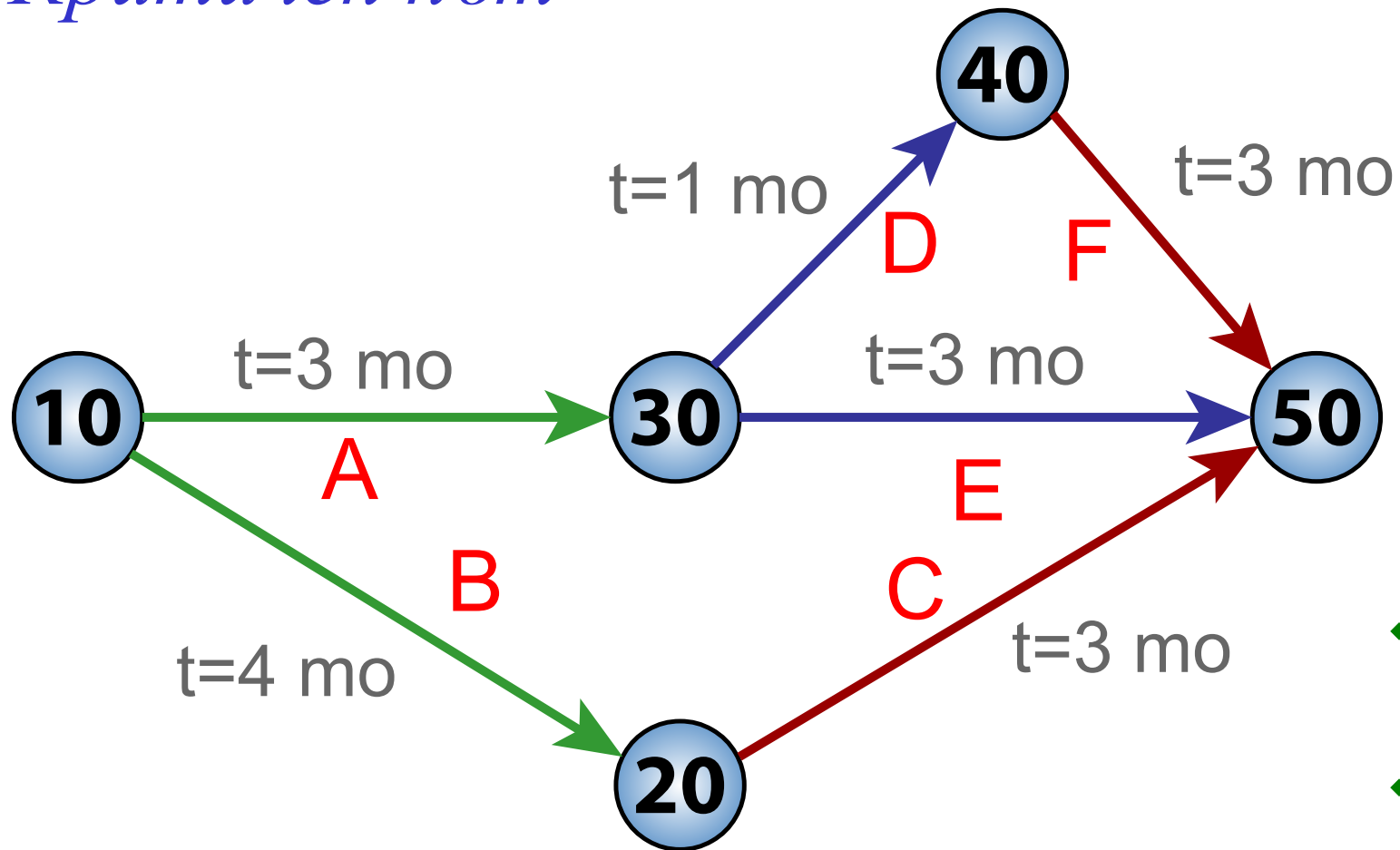


# Как се реализира?

- ❖ PThreads;
- ❖ Threading Building Blocks (TBB)
- ❖ Cilk++
- ❖ OpenMP
- ❖ MPI, MPI-2
- ❖ OpenCL, CUDA
- ❖ ...

# *Критичен Път*

# Критичен път



- ❖ “Най-тежкия” път;
- ❖ Тук имаме два: ADF и BC;



# *Анализ на Паралелни Алгоритми*

# Пример 1

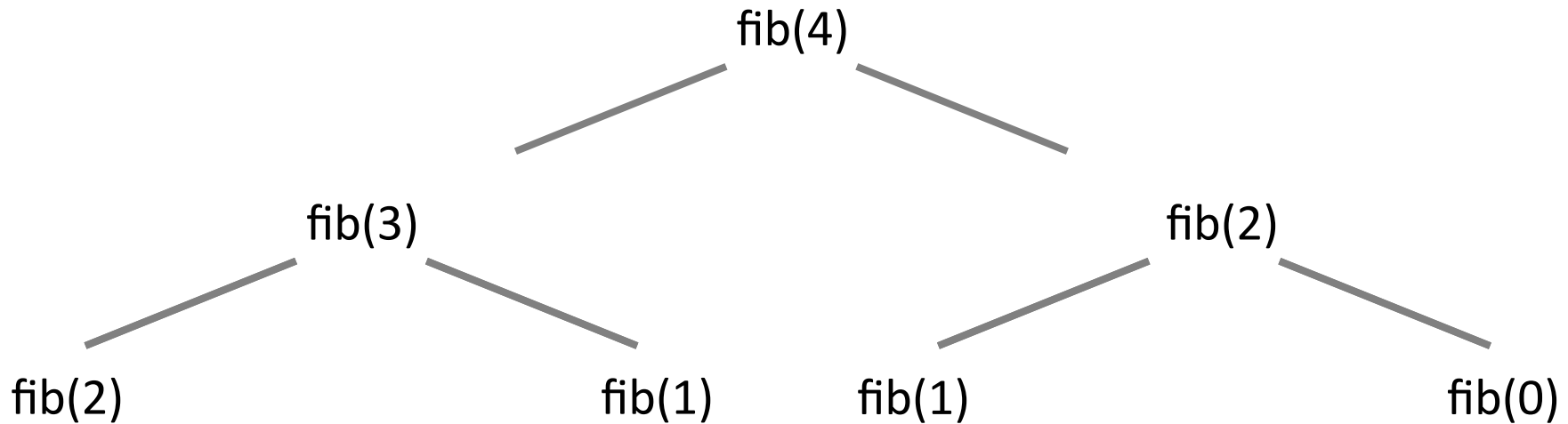
```
int fib(int n) {  
    if (n < 2) return n;  
  
    int n1, n2;  
  
    #pragma omp task shared(n1)  
    n1 = fib(n - 1);  
    #pragma omp task shared(n2)  
    n2 = fib(n - 2);  
  
    #pragma omp taskwait  
  
    return n1 + n2;  
}
```

Указание за компилатора за  
автоматично  
разпаралеляване на  
следващия statement;

Ако компилатора не поддържа автоматична паралелизация (OpenMP), то програмата се компилира както до сега



# Анализ на алгоритъм пресмятащ числата на Фибоначи (рекурсивен вариант)



# Важни характеристики

## ❖ $T_p$ – Времето за изпълнение на $P$ процесора

*Времето за изпълнение на алгоритъма на повече от един процесор може да съвпада с това за изпълнението на повече процесори, ако алгоритъма не е паралелизиран*

## ❖ $T_\infty$ – Време за изпълнение на т.н. критичен път

*Може да смятаме че критичния път е възможно най-бавния възможен път за изпълнение на алгоритъма. Това време се нарича още период (*span* или *depth*)*

## ❖ $T_1$ – Време за изпълнение на 1 процесор

*Без паралелни изпълнения, независимо дали програмата е паралелизирана или не. Това време се нарича още работа (*work*)*

## ❖ $P \cdot T_p$ – Пълното време загубено от всички процесори

*Това е времето за изпълнение и изчакване, загубено от всички процесори. Това време се нарича още разход (*cost*)*

# Важни характеристики

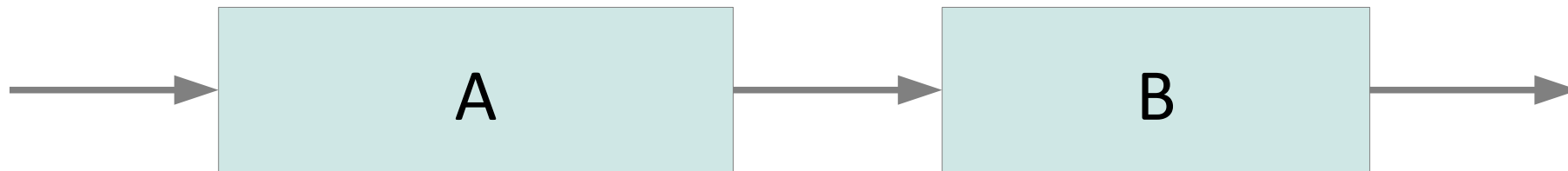
- ❖ Закон за Работата (work law):

$$T_p \geq T_1/P \text{ или } P \cdot T_p \geq T_1$$

- ❖ Закон за Периода (span law):

$$T_p \geq T_\infty$$

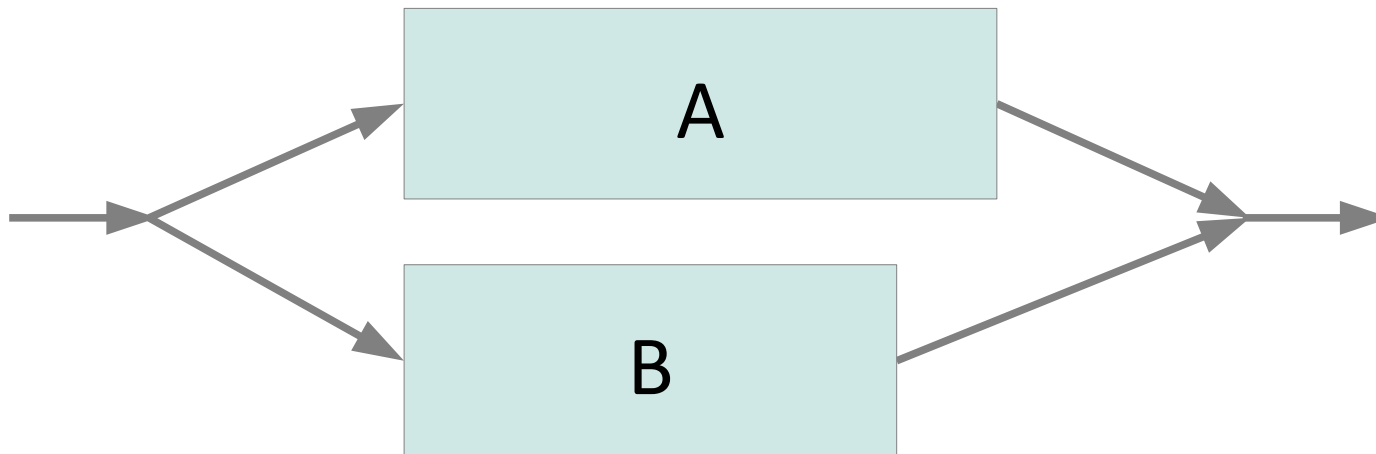
# Времена на паралелен алгоритъм



$$T_1(A \cup B) = T_1(A) + T_1(B)$$

$$T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$$

# Времена на паралелен алгоритъм



$$T_1(A \cup B) = T_1(A) + T_1(B)$$

$$T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$$

# *Ускорение и Забавяне*



# Фактор на ускорение (*Speedup*)

**Фактор на ускорение** на  $P$  процесора се нарича

$$S_p = T_1 / T_p$$

- ❖ Линейно при  $T_1/T_p = k \cdot P$
- ❖ Перфектно линейно при  $T_1/T_p = P$
- ❖ Супер линейно при  $T_1/T_p > P$

*Невъзможно? На практика е възможно поради наличие на Кеш паметта и други фактори...*

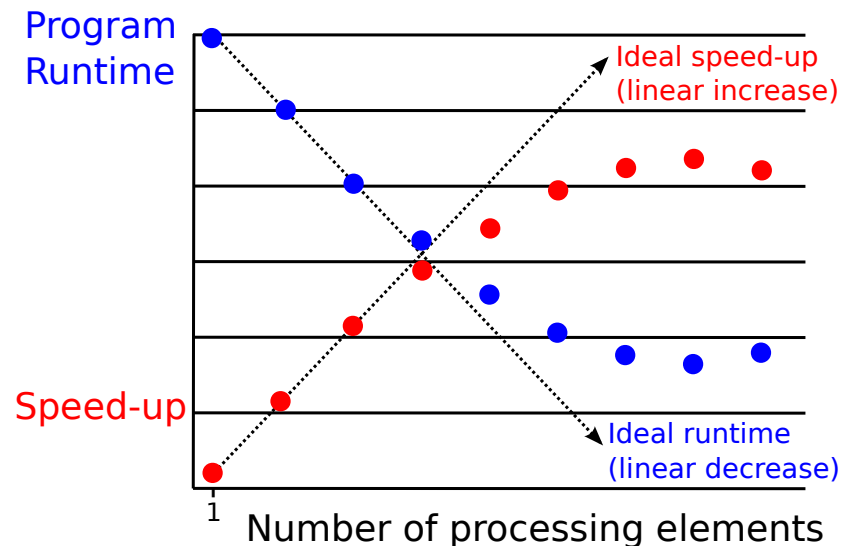
# Скалируемост (*Scalability*)

Алгоритми, които има линеен фактор на ускорение се наричат **скалируеми**.

# Забавяне (*Parallel slowdown*)

**Паралелното забавяне** е феномен в паралелните изчисления, при което паралелизирането на алгоритъм над определена точка кара програмата да работи по-бавно (отнема повече време, за да се изпълни до завършване).

Това обикновено се дължи на тесни места или проблеми в комуникацията.



# *Коефициент на Ефективност*

# *Ефективност (Efficiency)*

**Коефициент на ефективност се нарича**

$$S_p / P$$

# *Коефициент на Паралелизъм*

# Паралелизъм (Parallelism)

Коефициент на паралелизъм се нарича

$$T_1 / T_\infty$$

❖ Ако  $T_1 / T_\infty < P$ , то  $T_1 / T_P < T_1 / T_\infty < P$

# Паралелизъм

❖ За  $\text{fib}(4)$  имаме Паралелизъм  $\approx 17/8 = 2.125$

*Използването на повече от 2 процесора няма да даде съществено подобрение*

❖ За умножение на матрици  $1000 \times 1000$  имаме  
Паралелизъм  $\approx 10^6$

*В зависимост от реализацията паралелизма може да достигне и  $10^7$*

❖ За един нормален RayTracer (при сцена  $1000 \times 1000$  пиксела)  
имаме Паралелизъм  $\approx 10^6$

*Много груба оценка и то за не рекурсивен RayTracer. На практика може да е и много по-голяма*



# *Коефициент на Отпуснатост*

# Отпуснатост (Slackness)

**Коефициент на отпуснатост се нарича**

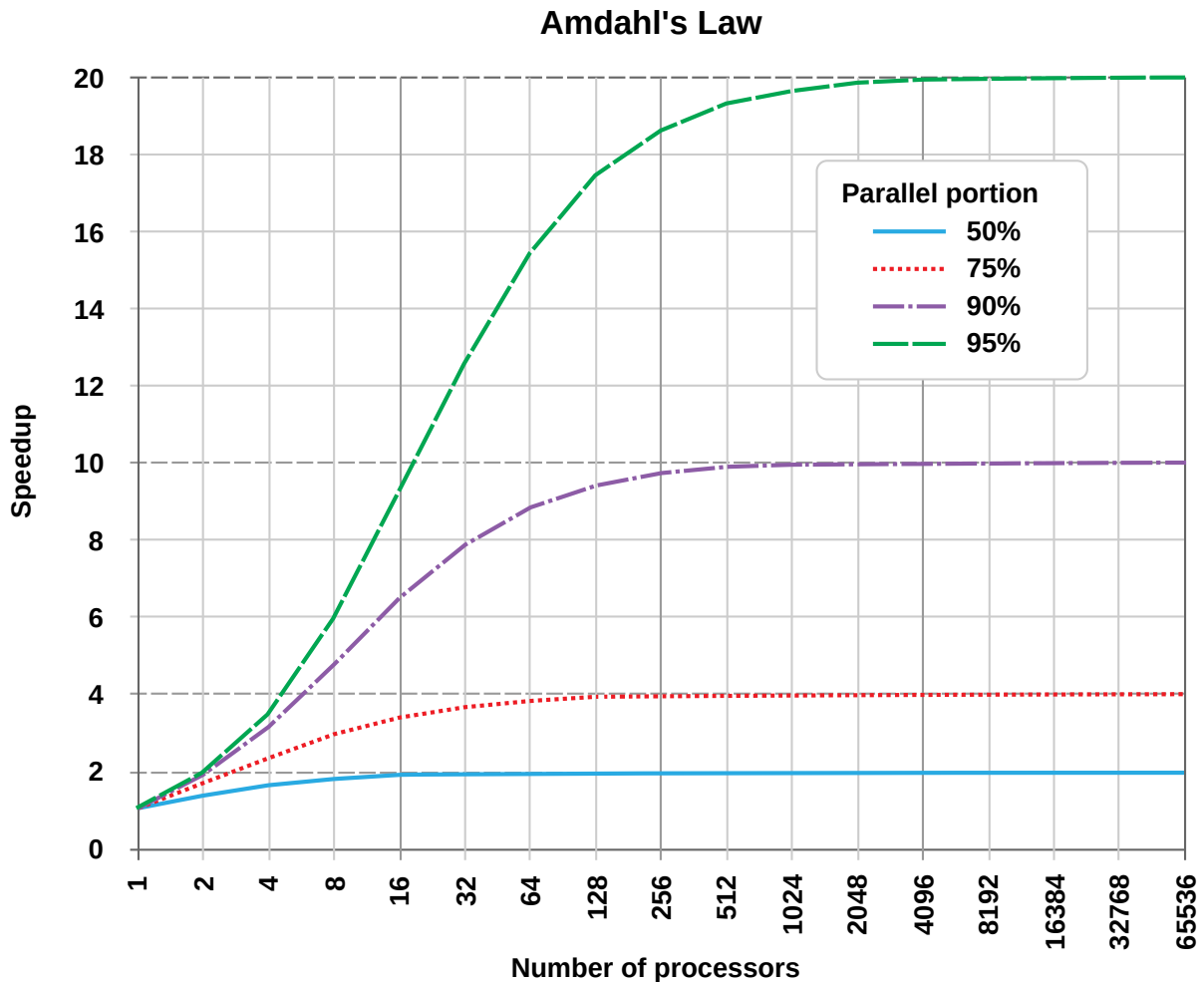
$$T_1 / (P \cdot T_\infty)$$

- ❖ Ако този коефициент е по-малък от 1, то перфектното линейно ускорение е невъзможно.

# *Закон на Амдала*

*Amdahl*

# Закон на Амдала /Amdahl/ (1967)



$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

$S_{latency}$  е теоретичното ускорение;

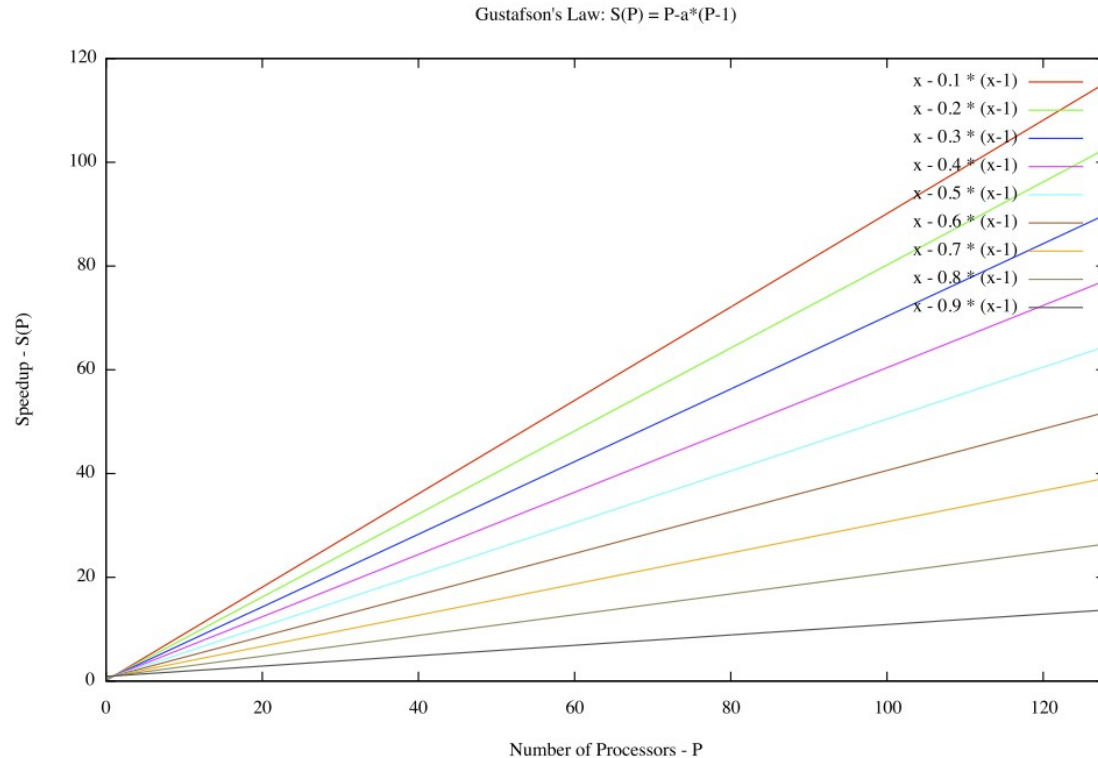
$S$  е ускорението на частта от задачата, която се възползва от тези паралелни ресурси;

$P$  е частта на времето за изпълнение, която се възползва от паралелните ресурси;

# *Закон на Густавсон*

*Gustafson*

# Закон на Густвсон /Gustafson/ (1988)



$$S_{latency}(s) = 1 - p + s \cdot p$$

$$S(sf) = N + (1 - N) \cdot sf$$

S е ускорението в латентността на изпълнението на частта от задачата, която се възползва от паралелните ресурси;

P е процентът на натовареността на изпълнението на цялата задача по отношение на частта, която се възползва от подобряването на ресурсите на системата преди подобряването.

# *Метрики на Karp–Flatt*

# Метрика на Karp–Flatt (1990)

Метриката на Karp-Flatt измерва паралелизацията на кода в паралелните много процесорни системи.

Нека е дадено паралелно изчисление показващо ускорение  $\psi$  на  $p$  процесора, където  $p > 1$ , експериментално определената серийна фракция  $e$  е дефинирана от метриката на Karp-Flatt:

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

*Колкото е по-малко  $e$ , толкова е по-добра паралелизацията*



# Метрика на Karp–Flatt (1990)

Това може да се запише и като:

$$T(p) = T_s + \frac{T_p}{p}$$

където:

$T(p)$  е времето за изпълнение на кода на  $p$  процесора;

$T_s$  е времето отнето от серийната фракция (не паралелната част);

$T_p$  е времето на паралелната част изпълнена на един процесор;

$p$  е броя на процесорите;



# *Детайлност/Прецизност*

*(Granularity)*

# Детайлност/Прецизност (*Granularity*)

При паралелното програмиране, **детайлност** е качествена мярка на съотношението на изчисление и комуникация. Детайлността бива:

- ❖ Груба (Coarse) – сравнително голям обем от изчислителна работа се извършва между комуникационните събития;
- ❖ Финна (Fine) – сравнително малко количество изчислителна работа се извършва между комуникационните събития;

