



Паралелно Програмиране

Въведение

доц. д-р Александър Пенев

Какво е Паралелно Програмиране?

Въпрос с Прост Отговор:

*Използването на повече от един процесор/процес
(обикновено едновременно),
за изпълнението на дадена задача.*

И ...

Сложно Съдържание



Въведение

Теми разглеждани в курса

1. Въведение в паралелното програмиране.

Пример. Кратка история. Базови понятия и концепции. Паралелни архитектури. Модели за паралелно програмиране. Възможни проблеми. Пример – критична секция. Практически проект.

2. Съвременни паралелни архитектури. Класификация на Флин (Flynn). CPU, GPGPU, HSA, Високо производителни компютри (HPC).

Класификация на Flynn (SISD SIMD SIMT MISD MIMD). Скаларни/Pipelined и Суперскаларни процесори. SIMD инструкции. Dataflow architecture. Vector processor. Multiprocessor (symmetric/asymmetric). CPU/GPGPU. HSA. HPC.

3. Памет. Йерархия на паметта. Архитектури на паметта на паралелните компютри.

Памет. Кеш. Архитектури (shared, distributed, distributed shared, UMA, NUMA, COMA), Shared-Nothing Architecture и др. Massively parallel computer (GRID Системи, Компютърни Кълстъри, Силно паралелни процесорни масиви).

4. Видове паралелизъм. Векторизация. Задачи – фибри, нишки, процеси. Видове многозадачност.

Видове паралелизъм: Bit Level; Instruction (ILP); Task; Data; Memory. Векторизация. Векторизация на цикли. Задачи – фибри, нишки, процеси. Видове многозадачност: Temporal; Simultaneous (SMT); Speculative (SpMT); Preemptive; Cooperative; Clustered Multi-Thread (CMT).

5. Теоретични аспекти на паралелните алгоритми. Анализ на паралелни алгоритми.

Зависимости на данните, структурата и контрола (Data dependency, Data, Structural and Control Hazards).

PRAM модел. PEM модел. Разпаралеляване. Анализ на паралелни алгоритми. Критичен път. Закони на Amdahl. Закон на Gustafson. Метрики на Karp–Flatt. Забавяне и Ускорение. Кофициенти и метрики. Granularity...



Теми разглеждани в курса

6. Класически алгоритми и проблеми. Алгоритъм на Декер. Проблеми при паралелните алгоритми: Мъртва хватка, жива хватка, трудна скалируемост, глад за ресурси, съперничество и др. Producer-Consumer.

Задача за „Вечерящите философи“ и др. Алгоритъм на Декер. Задача за „Тримата пушачи“. Задача за „Спящият бърснар“. Deadlock, Livelock, Parallel slowdown, Race condition, Software lockout, Scalability, Starvation, Convoying, Contention; Deterministic algorithms; Embarrassingly parallel; Producer-Consumer.

7. Модели за паралелно програмиране. Координация в паралелните алгоритми.

Модели за ПП: Shared Memory model; Threads model; Message Passing model; Implicit interaction model; Data Parallel model. Видове координация/синхронизация в паралелните алгоритми: Barrier; Locks; Semaphores; Mutexes, ...

8. Дизайн на паралелни програми.

Проблем и решения. Декомпозиция на данни и алгоритми. Видове комуникация между подзадачите. В/И – проблеми и решения. Fork-Join. Map-Reduce. Hotspots и Bottlenecks. Не блокиращи алгоритми и структури данни. Zero-Copy. Read-Copy-Update (RCU/COW). Транзакционна памет.

9. Езици и Библиотеки (API) за паралелно програмиране.

Езици за ПП. Примери. Библиотеки: C++11 STL (Futures, Promises, Threads, ...); C# (Threads, TPL, Tasks, Furures, PLINQ, async, await, yield, ...); Java (Threads, Locks, Atomic, Futures, Streams, ...); OpenMP (Fork-Join, ...); MPI/MPI-2 (Комуникация, синхронизация, паралелен В/И, редукции, ...); POSIX Threads, Boost.Thread, TBB; CUDA (Примери, PyCUDA, ...); OpenCL (); OpenHMPP, OpenACC, C++ AMP (Коделети, kerneli, #pragma, GPU, ...).

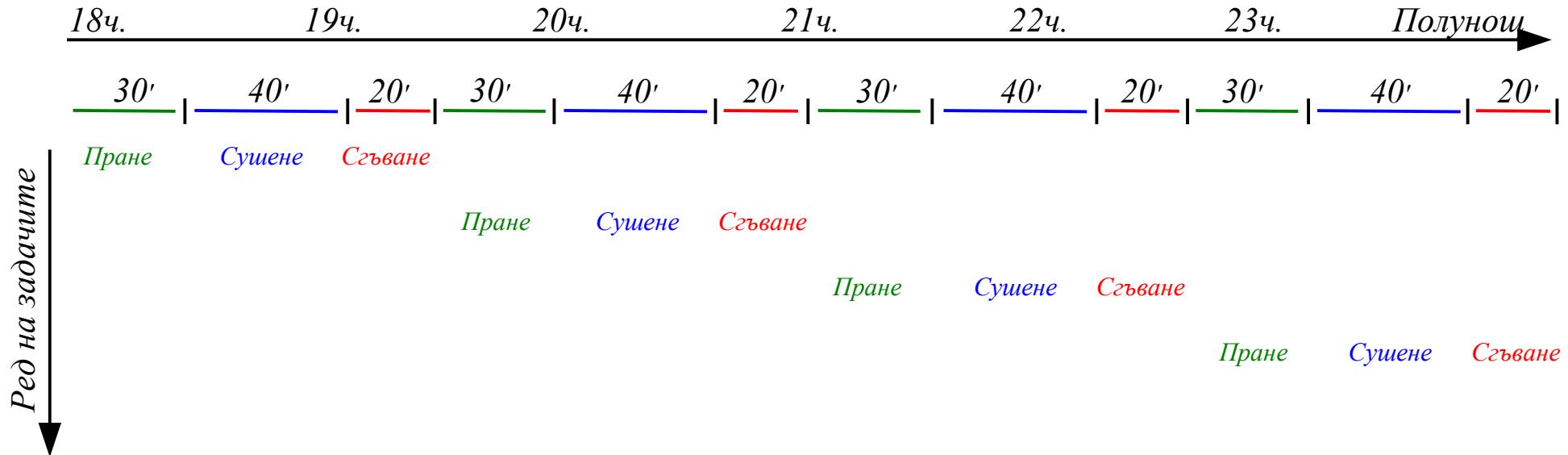
10. Бъдеще на паралелните архитектури и програмиране.

GPGPU, TPU, FPGA, ASIC, NPU, QPU – Квантов паралелизъм и др.



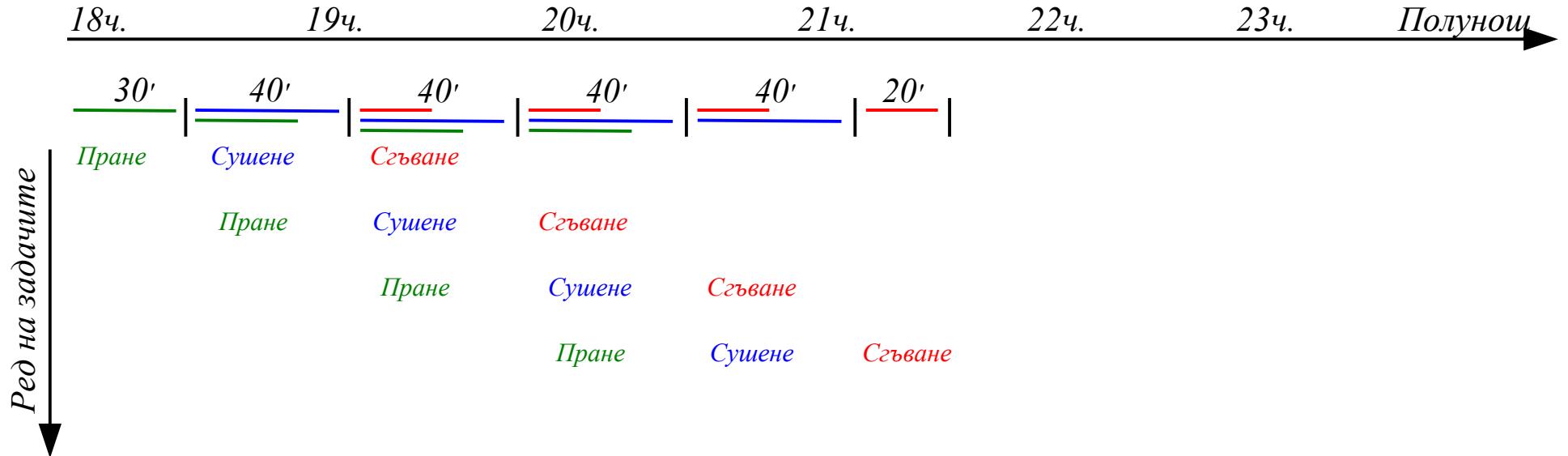
Пример

Последователно пране



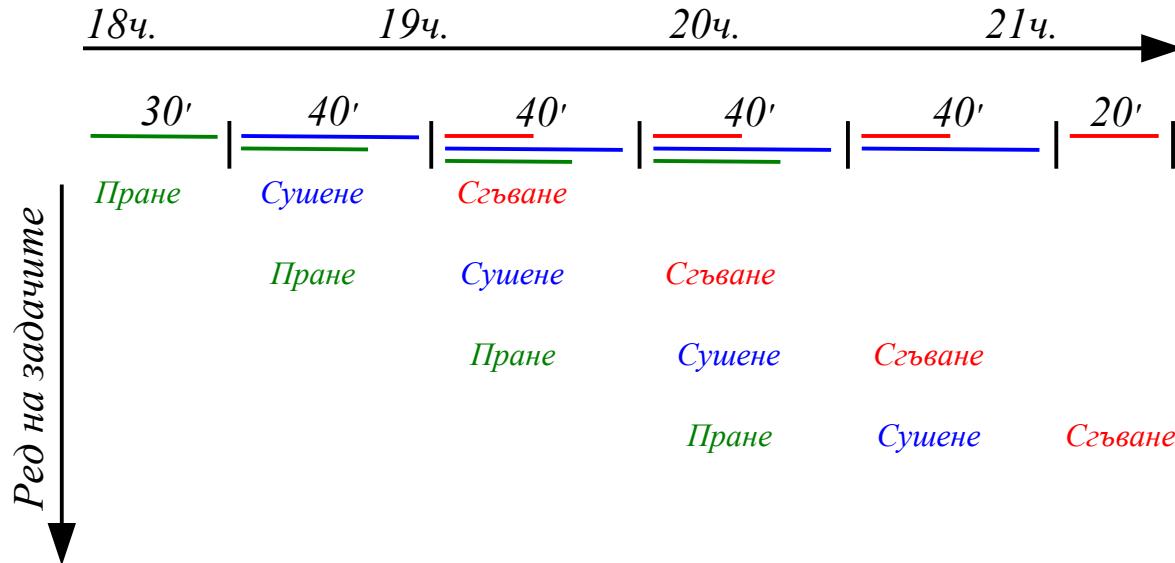
- Прането отнема 30 мин;
- Сушенето отнема 40 мин;
- Сгъването отнема 20 мин;
- Последователното пране отнема 6 часа за 4 купа дрехи;

Конвейерно пране



- Конвейерно означава задачата да се започва колкото се може по-скоро;
- Конвейерното пране отнема 3.5 часа!

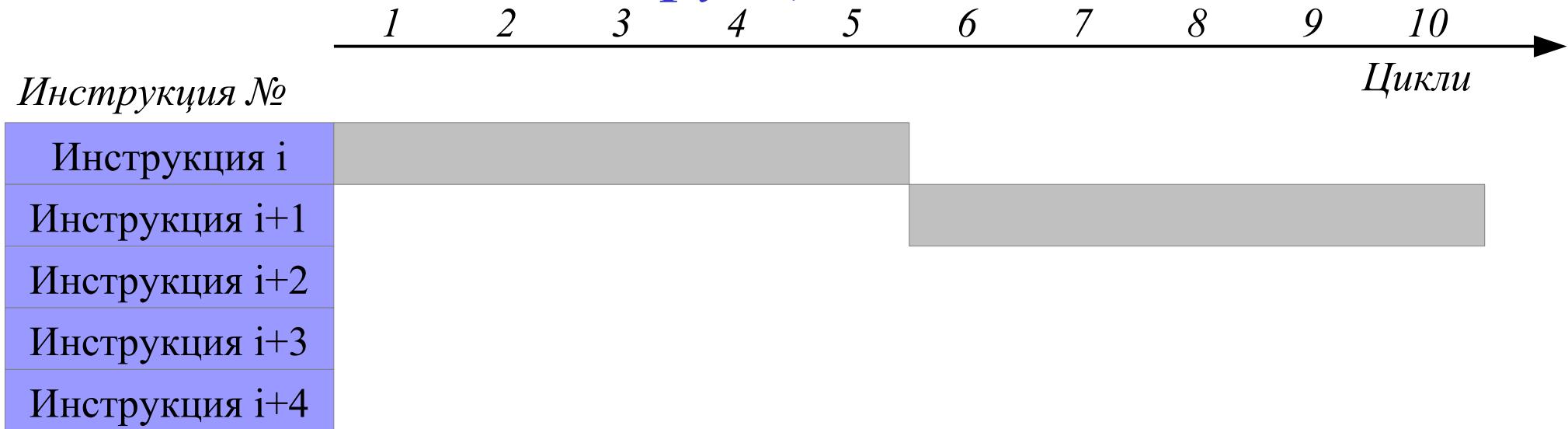
Изводи от конвейерното изпълнение



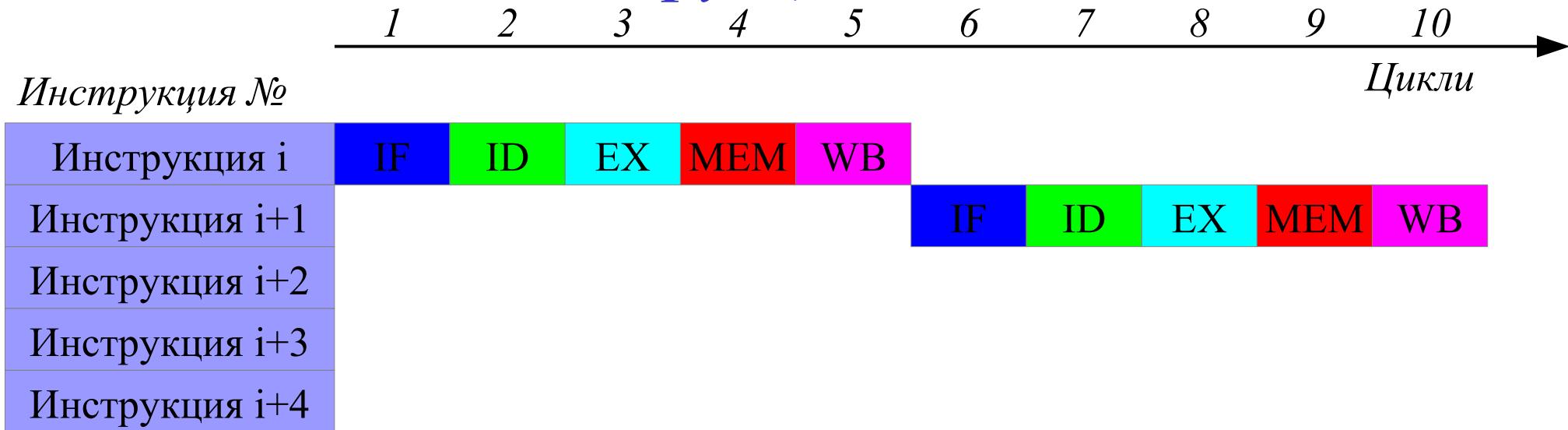
- Конвейерното изпълнение не намалява латентността на отделните задачи, а само пропускателната способност на цялото задание;
- Ефективността на конвейера е ограничена от най-бавната операция;

- ❖ Множествено задачи се изпълняват едновременно когато използват различни ресурси;
- ❖ Възможното ускорение е броя стъпки в конвейера;
- ❖ Небалансираните дължини на стъпките намаляват ускорението;
- ❖ Времето за напълване на конвейера и времето на изпразване намаляват ускорението;
- ❖ Stall при наличие на зависимости;

Изпълнение на инструкции

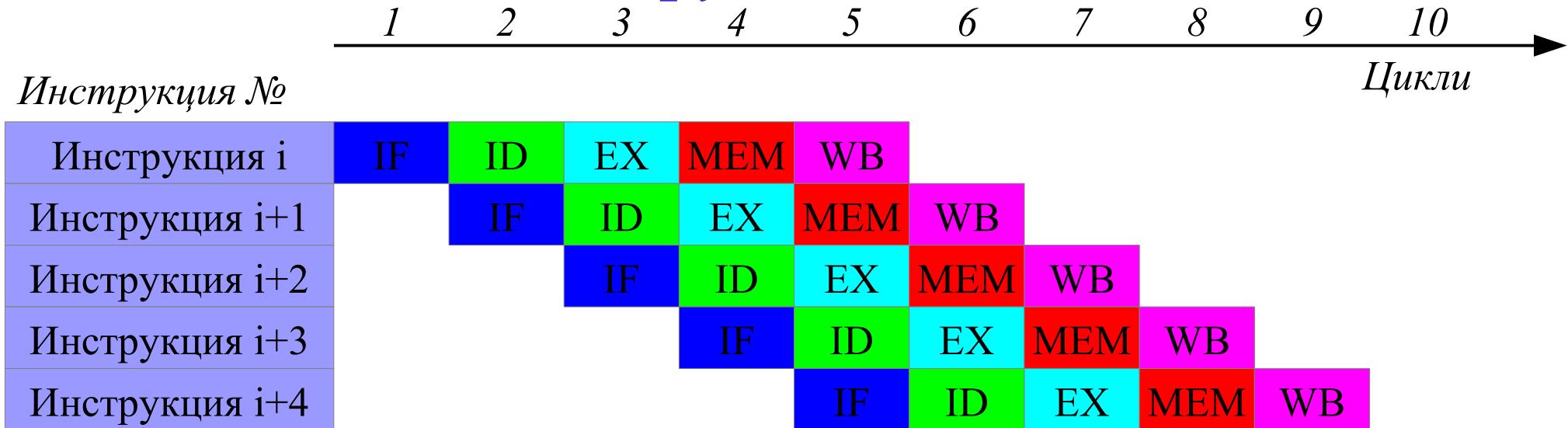


Изпълнение на инструкции



- IF – Instruction Fetch
- ID – Instruction Decode
- EX – Execution
- MEM – Memory Access
- WB – Write Back

Изпълнение на инструкции



- IF – Instruction Fetch
- ID – Instruction Decode
- EX – Execution
- MEM – Memory Access
- WB – Write Back

Кратка История



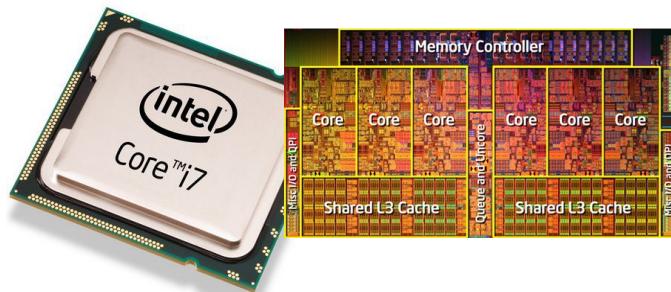
Защо?

Преди (70-те години на миналия век)



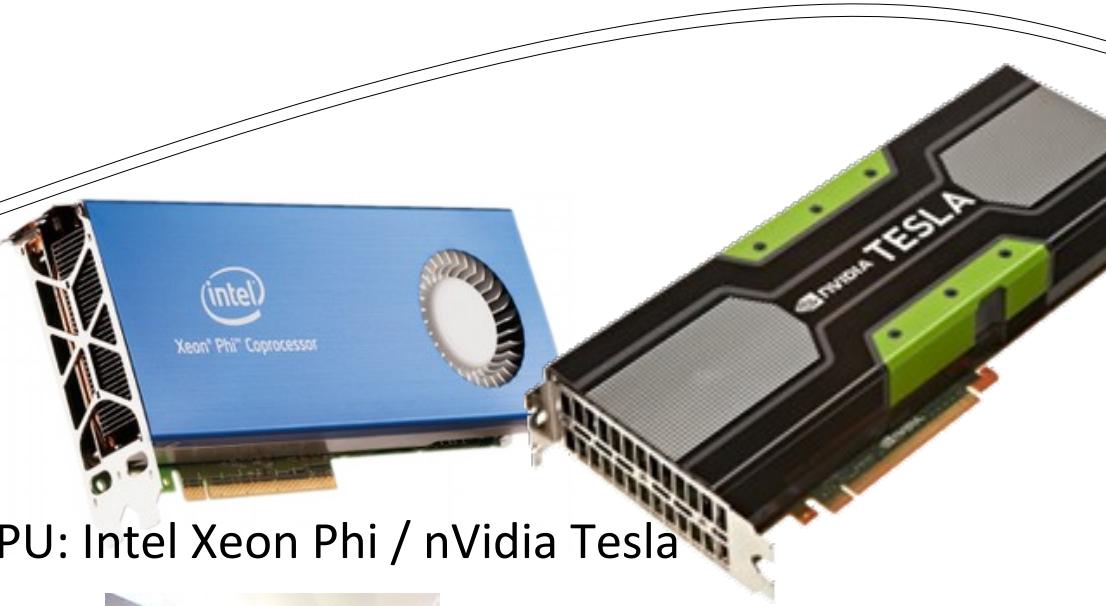
Intel 8086

Сега (2020 г.)



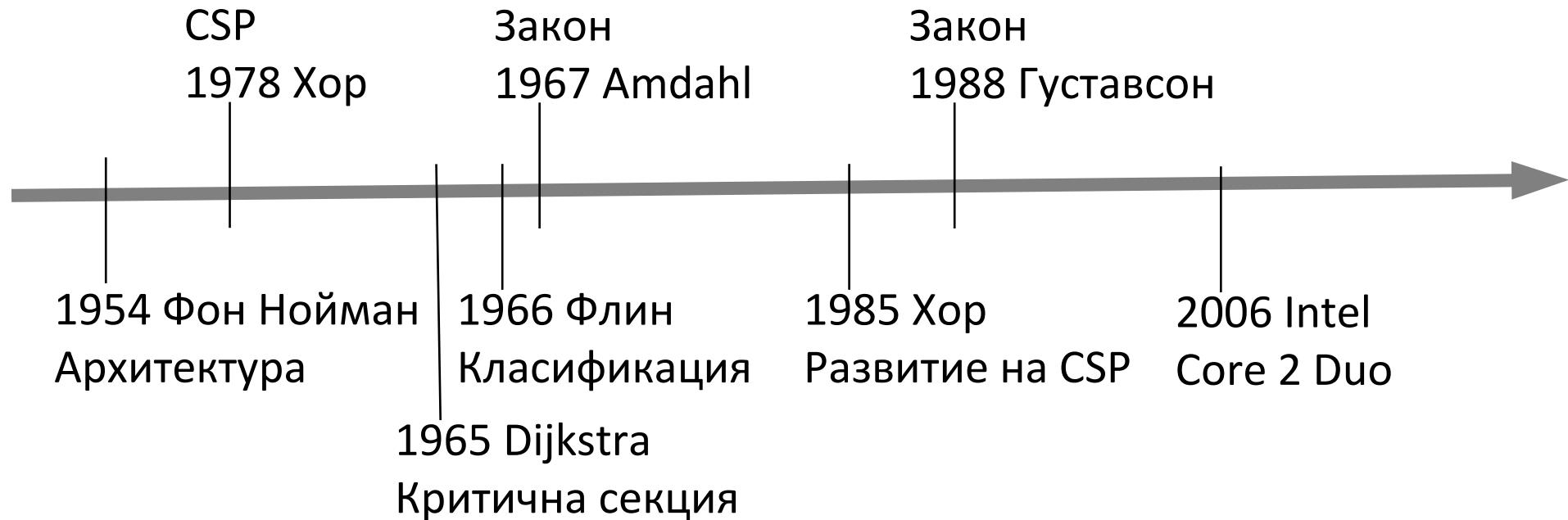
Intel i7 Процесор

GPGPU: Intel Xeon Phi / nVidia Tesla

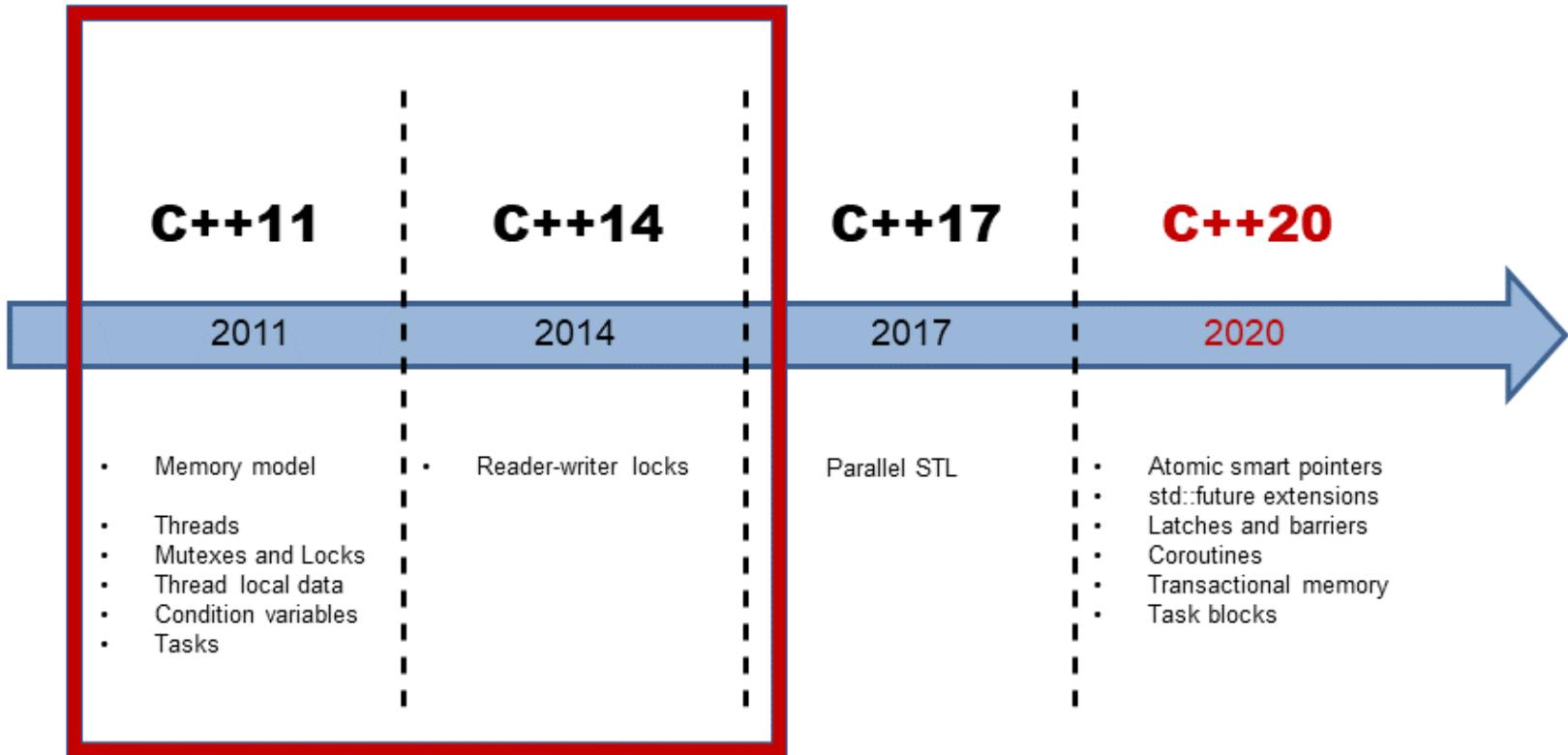


Data Centers

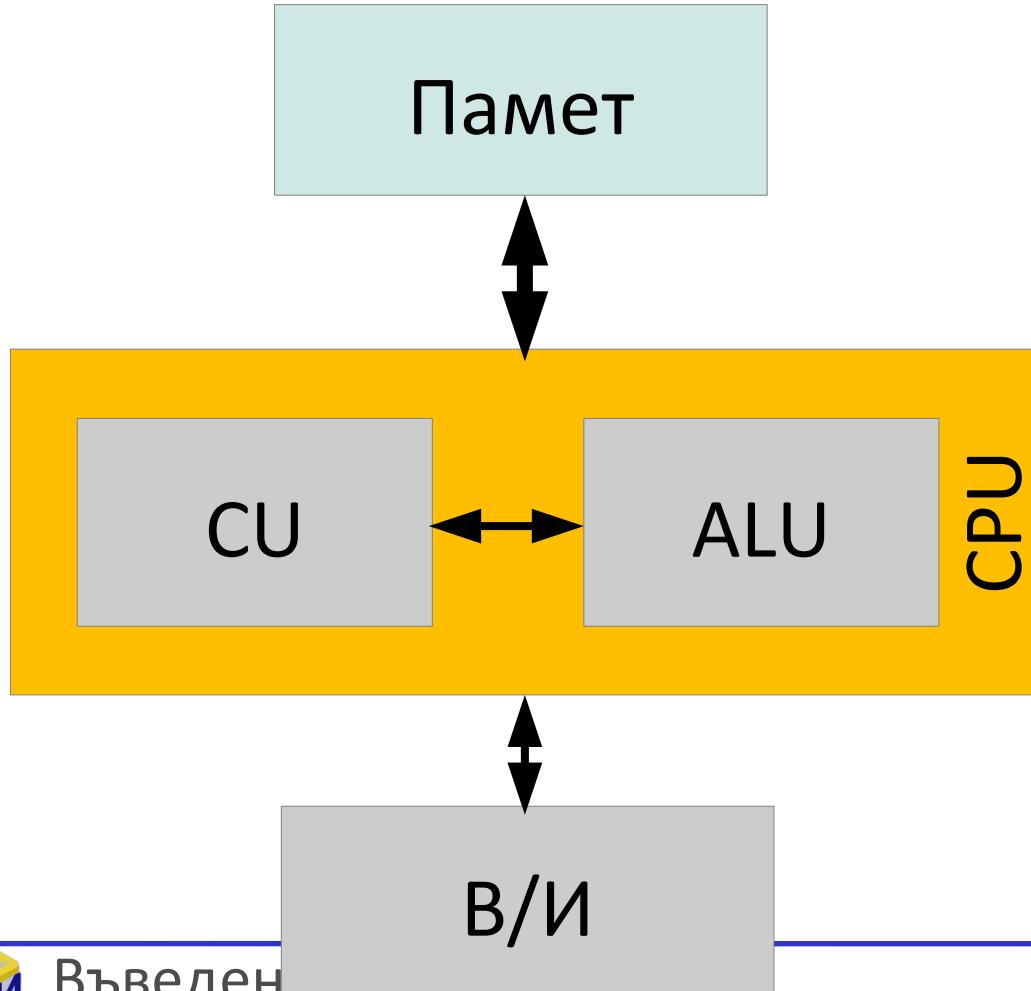
Времева линия



Времеева линия (2)

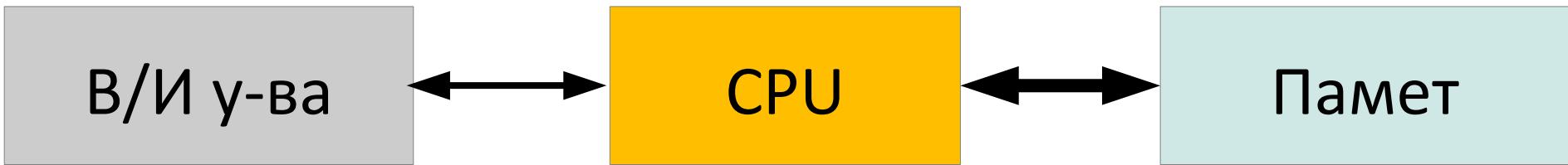


Фон Нойманова архитектура (1945)

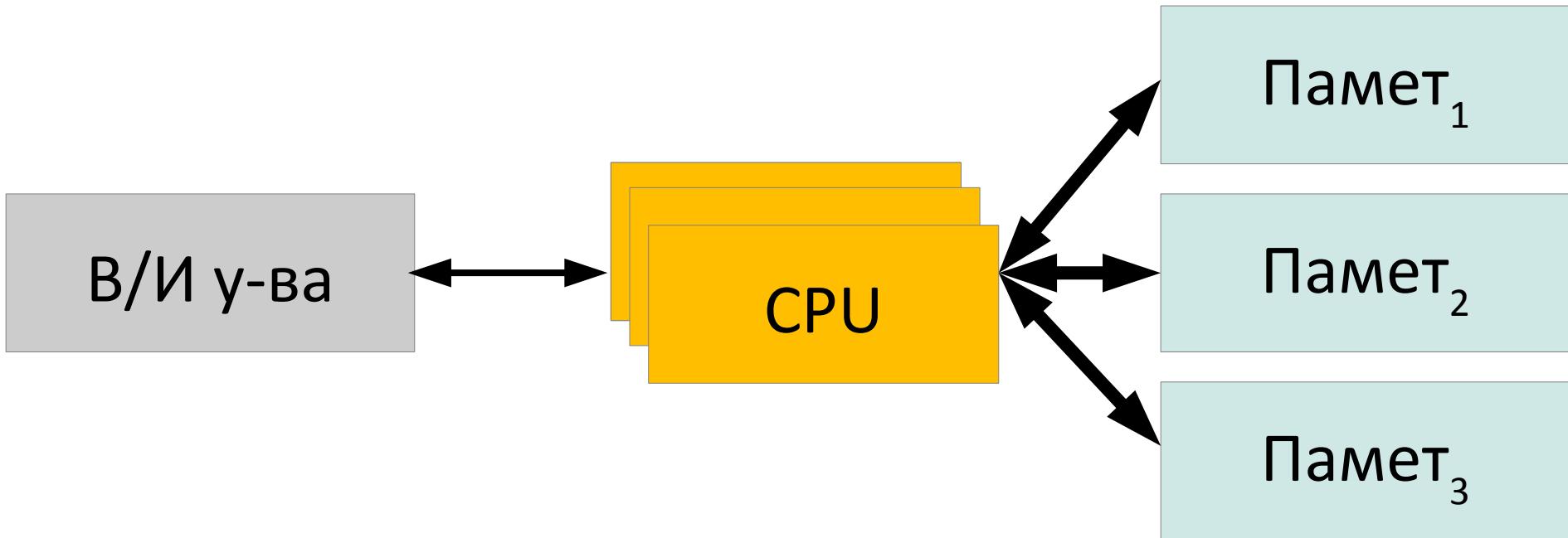


- ❖ Данните и Програмата се съхраняват в паметта;
- ❖ Контролното у-во (CU) извлича инстр. и данните от паметта, декодира ги и след това **последователно** координира изпълнението на операциите;
- ❖ Аритметичното и логическо у-во (ALU) извършва базовите аритметични операции;

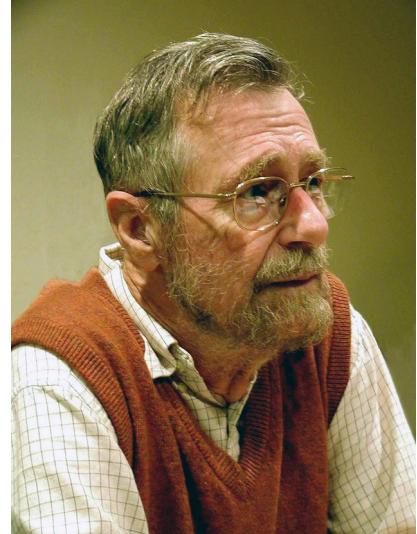
Класическа архитектура на компютър



Съвременен вариант на архитектурата



Дайкстра /Dijkstra/ (1965)



- ❖ За пръв път описва и дава име на критичните региони в паралелните алгоритми (Критична секция);
- ❖ Описва семафорите и проблема за „вечерящите философи“ (1968);
- ❖ Въвежда понятието “зашитени команди” (1975);

сър Тони Хор /Hoare/ (1978 и 1985)



- ❖ Работата му по CSP (Communicating Sequential Processes) – описва шаблоните за взаимодействие на конкурентни системи;
- ❖ С тази си работа Хор става един от пионерите на теорията и практиката на паралелните системи;
- ❖ Това дава тласък на множество езици за паралелно програмиране като OCCAM, Go и др., както и на инструменти за анализ на паралелни системи и др.;

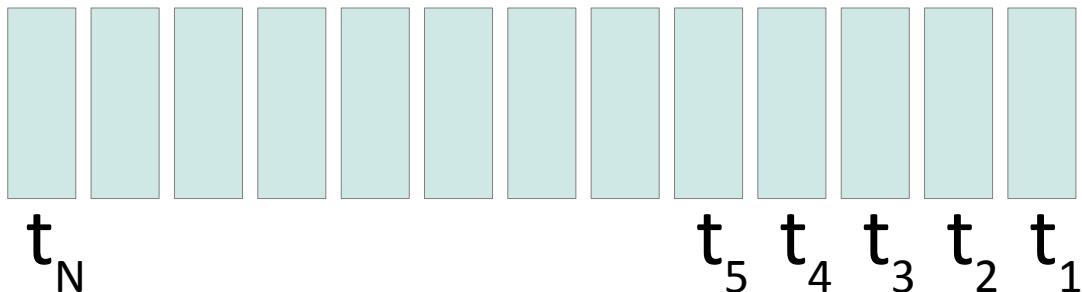
Базови Понятия и Концепции

Необходимост от Паралелни Архитектури (ПА) и Паралелно Програмиране (ПП)

- ❖ Физически ограничения пред скалируемостта;
скорост, пространство, памет, енергия, ...
- ❖ Решаване на по-големи проблеми;
- ❖ Решаване на проблеми по-бързо;
- ❖ Решаване на проблеми, които „не пасват“ на едно CPU;

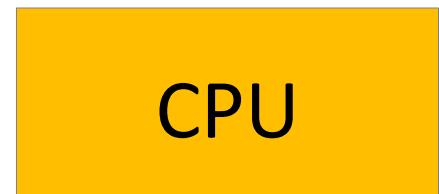
Не паралелно (серийно/последователно) изп.

Задача



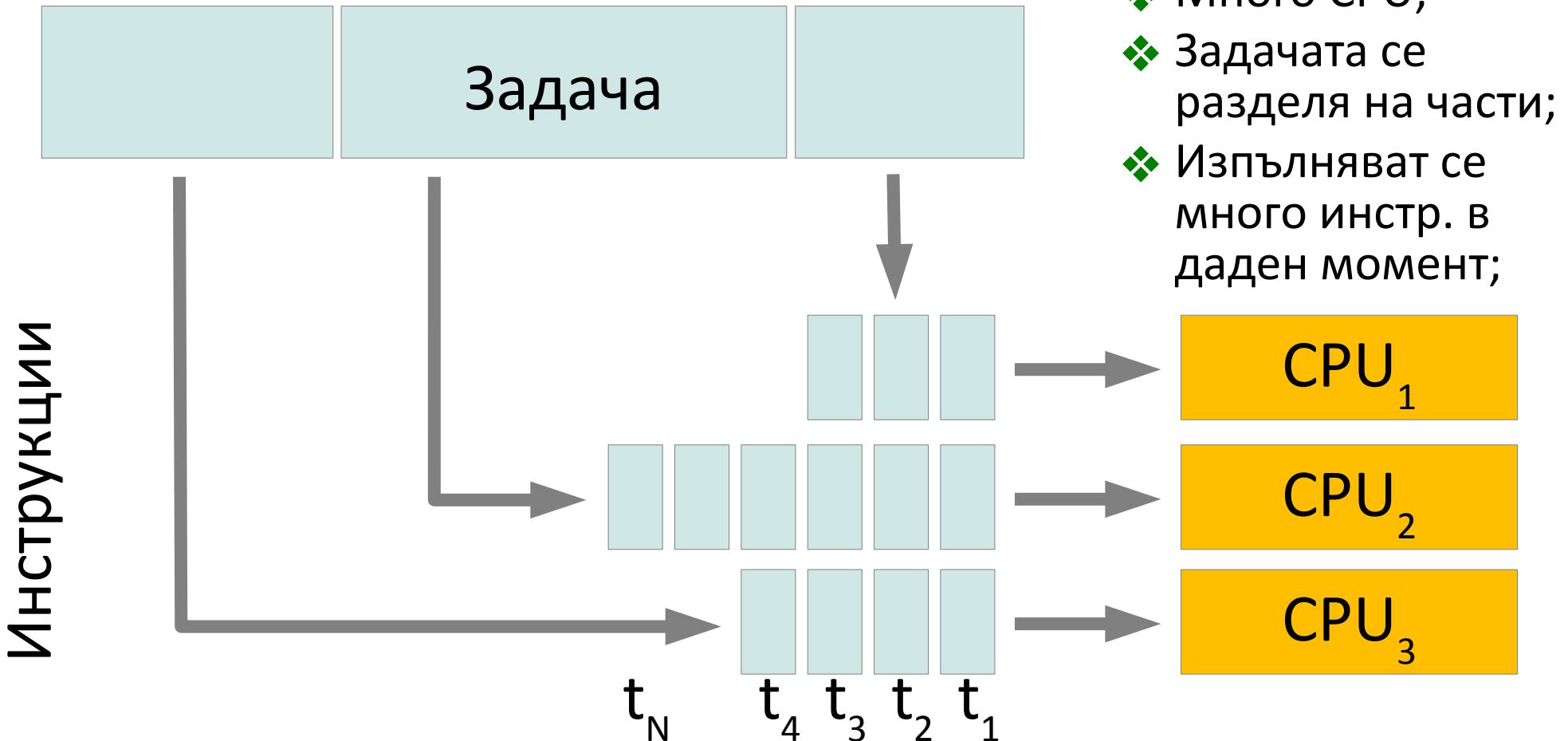
Инструкции

- ❖ 1 CPU;
- ❖ 1 Инструкция в даден момент;
- ❖ Инстр. се изпълняват една след друга;



CPU

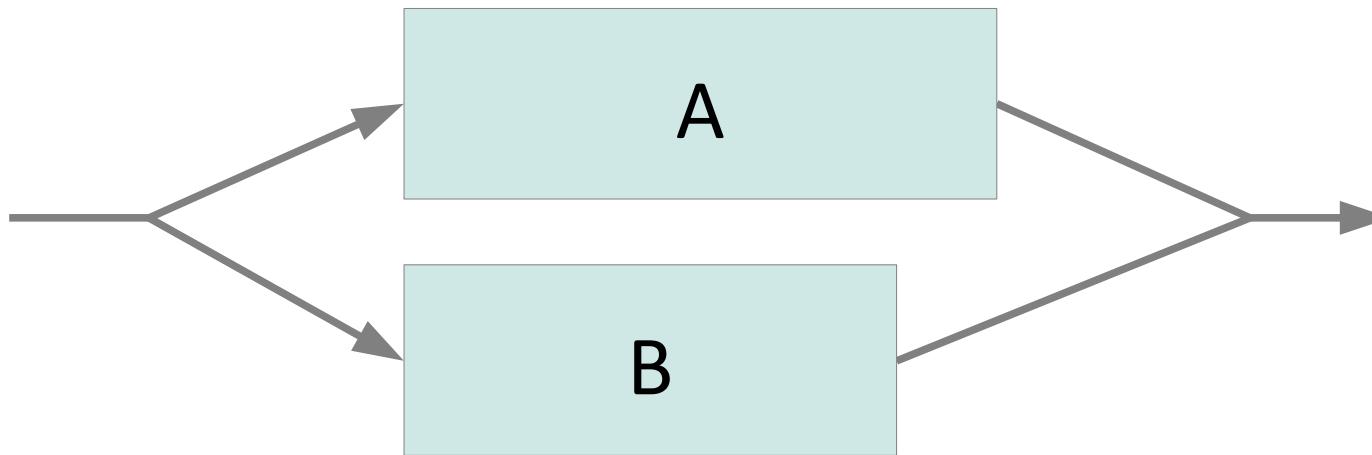
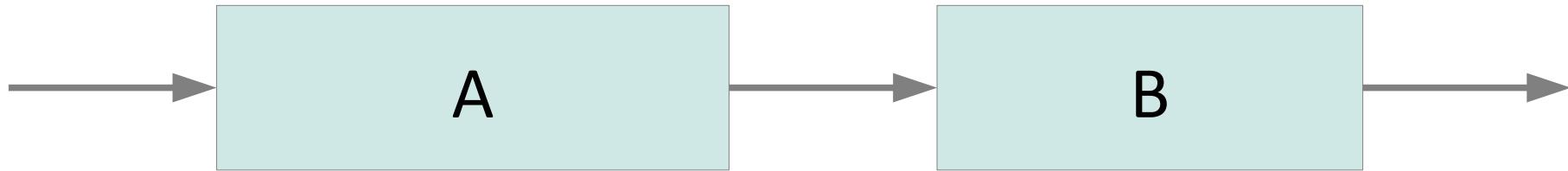
Паралелно изпълнение



Какво е паралелизация или разпаралеляване?

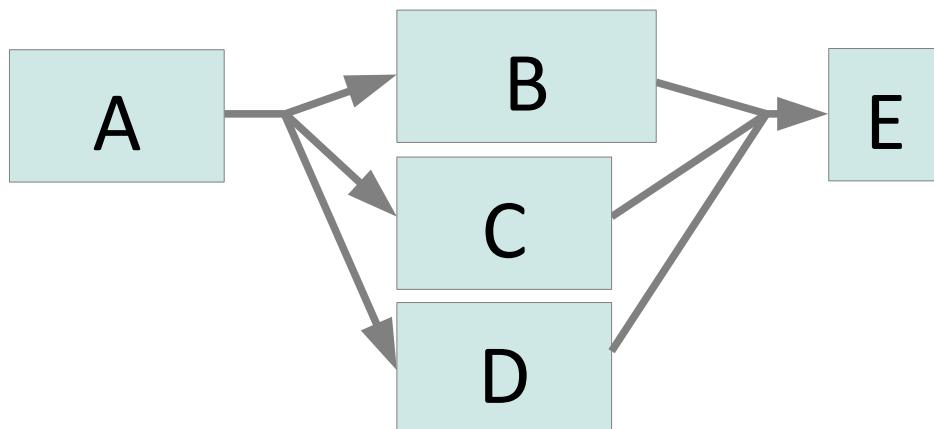
- ❖ Паралелизацията (parallelization) е процес на преобразуване на програма изпълняваща стъпките на алгоритмите си последователно, в програма изпълняваща ги (там където е възможно) паралелно/едновременно;
- ❖ Паралелното изпълнение може да е с използването на SIMD инструкции (векторизация), много нишки,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

Какво е паралелизация?

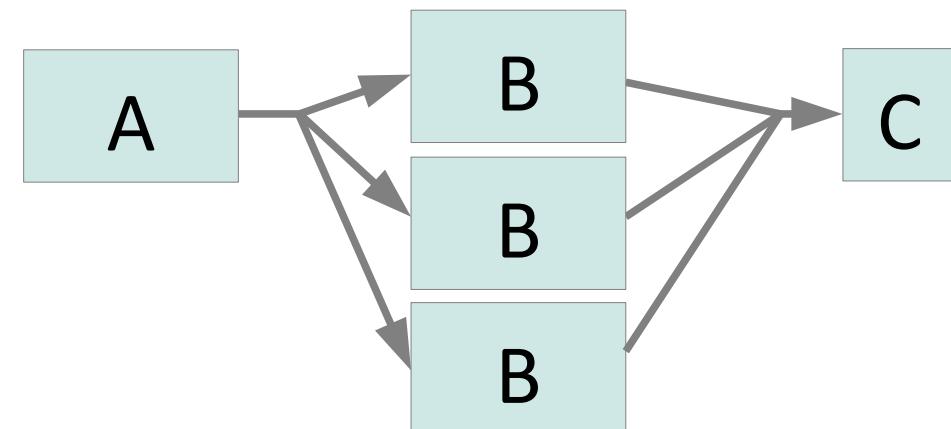


Функционален и Даннов Паралелизъм

- ❖ Функционален паралелизъм (Functional parallelism) – Всеки процесор работи върху част от проблема (задачата);
- ❖ Даннов паралелизъм (Data parallelism) – Всеки процесор извършва една и съща работа върху част от данните при решаването на проблема;

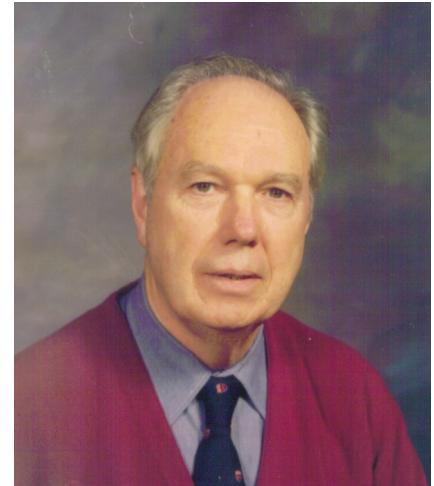


Функционален



Даннов

Класификация на Флин /Flynn/ (1966)



SISD

Старите Mainframe,
класическите PC и др.

SIMD

Повечето съвременни
PC, GPU и др.

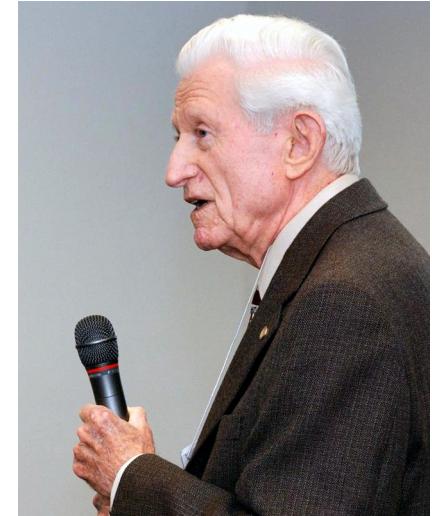
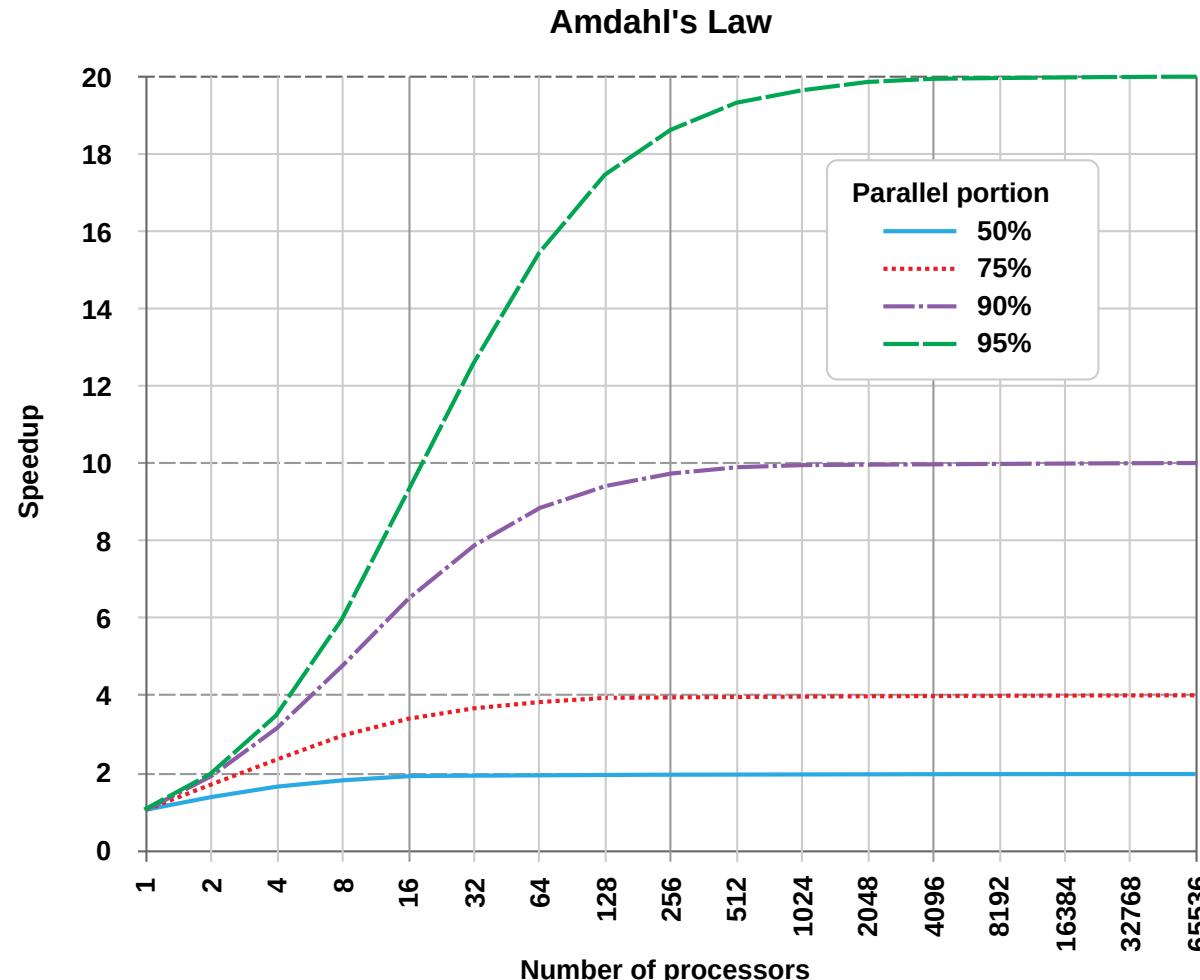
MISD

Използва
за системи защитени
от повреди,
С.тмр computer и др.

MIMD

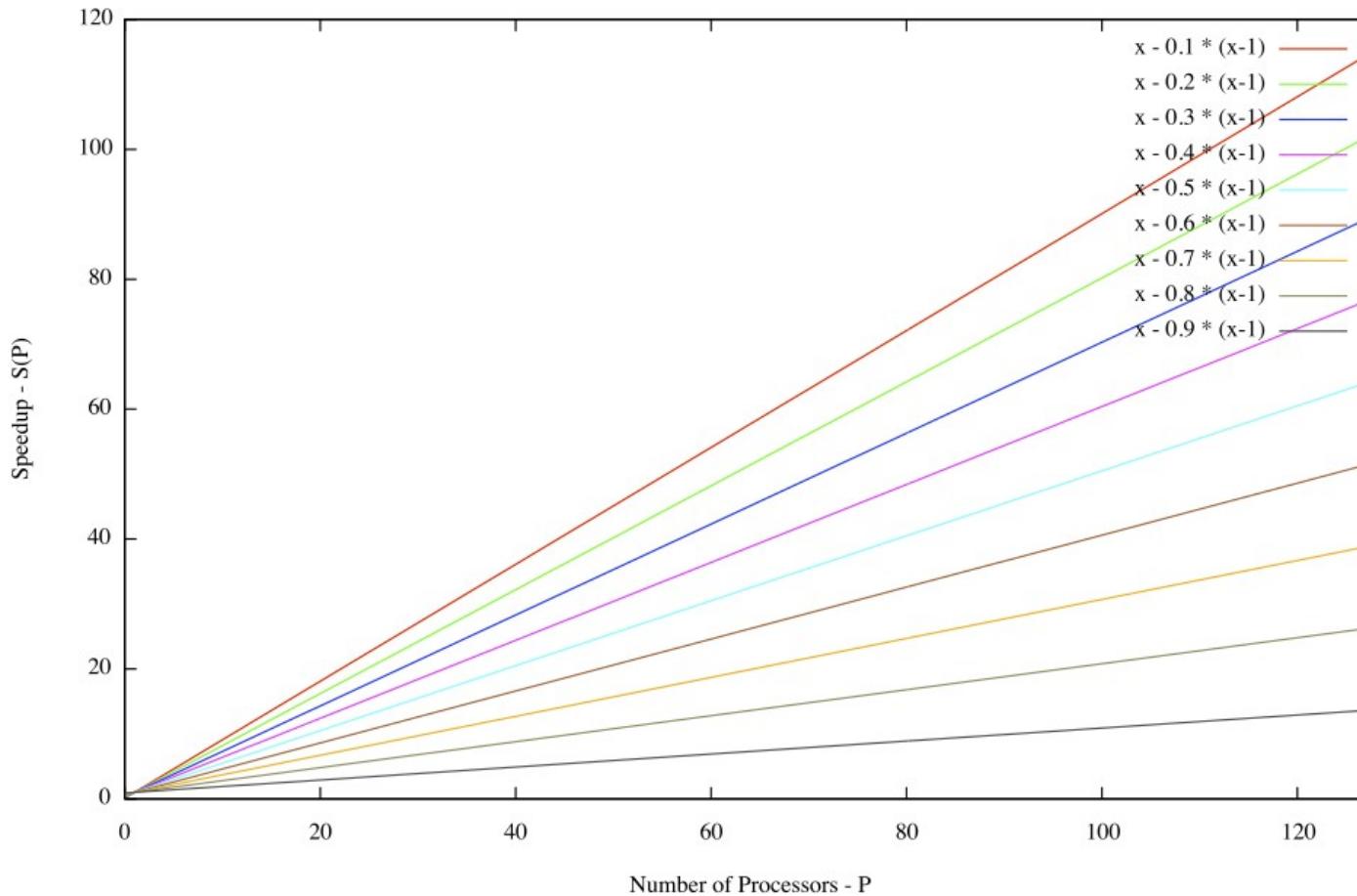
Много-ядрените
суперскаларни проц.,
разпределените
системи и др.

Закон на Амдала /Amdahl/ (1967)



Закон на Густфсон /Gustafson/ (1988)

Gustafson's Law: $S(P) = P \cdot a \cdot (P-1)$



Паралелни архитектури

❖ Споделена памет (Shared memory)

Много процесори могат да правят достъп до обща памет.

❖ Еднороден достъп до паметта (Uniform memory access – UMA)

Идентични процесори имат еднакъв достъп и еднакво време на достъп до паметта.

❖ Не-еднороден достъп до паметта (Non-uniform memory access – NUMA)

Не всички процесори имат еднакъв достъп и еднакво време на достъп до паметта.

❖ Cache only memory architecture (COMA)

Цялата памет на възлите се използва само като кеш.

❖ Разпределена памет (Distributed memory)

Изиска комуникация по мрежата за достъп до между процесорната памет.

❖ Хибридна Разпределено-Споделена памет (Hybrid Distributed-Shared Memory)



Модели за Паралелно Програмиране

❖ Споделена памет (Shared memory)

Задачите използват общо адресно пространство (обща памет), в която могат да пишат и четат асинхронно. Използват се различни механизми за защита и контрол на достъпа до общата памет

❖ Нишки (Threads)

Нишките са подпрограми в главната програма. Те комуникират помежду си през глобална памет. Примери за този модел са Posix Threads, OpenMP и др,

❖ Предаване на съобщения (Message Passing)

Множество от задачи, използващи своя собствена локална памет за изчисленията. Обмен на данни става чрез изпращане и получаване на съобщения. Трансфера на данни обикновено е кооперативен т.е. изпращащата страна трябва да има получател. Пример: MPI

❖ Данново паралелен (Data Parallel)

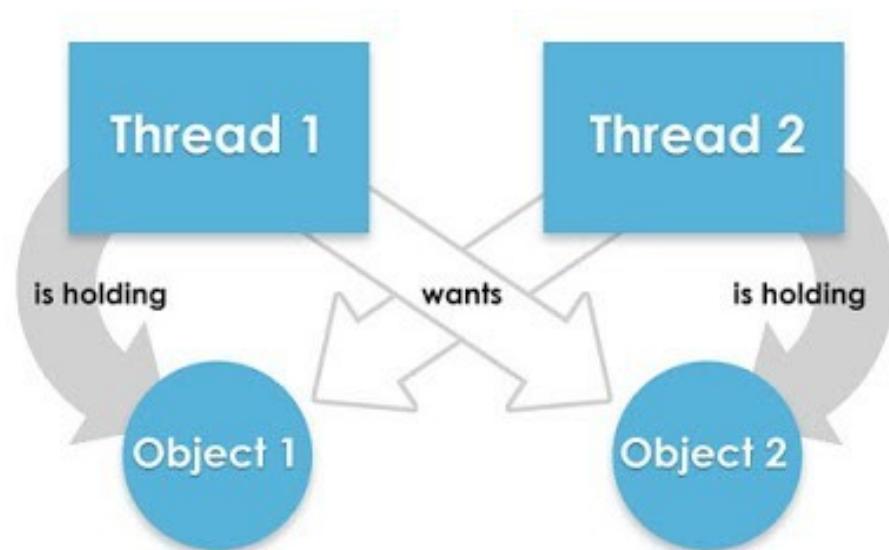
Множество от задачи, работещи колективно върху обща структура от данни. Всяка задача извършва едно и също действие със своята част от данните

❖ Хибриден (Hybrid)



Проблеми при ПП и Координация в Паралелни Алгоритми (ПАлг.)

- ❖ Мъртва хватка (Deadlock);
- ❖ Жива хватка (Livelock);
- ❖ Трудна скалируемост;
- ❖ Глад за ресурси;
- ❖ Съперничество;
- ❖ др.



Пример – критична скеция

```
void add(hashtable table, value v) {  
    int h = hash(v.key);  
    v.next = table[h].next;  
    table[h].next = &v;  
}
```

Може да възникне проблем
при паралелно изпълнение

```
h1 = hash(x.key);  
x.next = table[h1].next;  
  
table[h1].next = &x;
```

Thread 1

```
h2 = hash(y.key);  
  
y.next = table[h2].next;  
  
table[h2].next = &y;
```

Thread 2

Какъв е проблемът тук?



Пример – използване на критична секция

```
void add(hashtable table, value v) {  
    int h = hash(v.key);  
    mutex m;  
    m.lock();  
    v.next = table[h].next;  
    table[h].next = &v;  
    m.unlock();  
}
```

Критична секция

```
h1 = hash(x.key);  
m.lock();  
x.next = table[h1].next;  
table[h1].next = &x;  
m.unlock();
```

Thread 1

```
h2 = hash(y.key);  
  
m.lock();  
y.next = table[h2].next;  
table[h2].next = &y;  
m.unlock();
```

Thread 2

Няма проблем при $h1=h2$, обаче...

Какъв е проблема при тази реализация на критичната секция?



Пример – по-добро използв. на критична секция

```
void add(hashtable table, value v) {  
    int h = hash(v.key);  
    table[h].mutex.lock();  
    v.next = table[h].next;  
    table[h].next = &v;  
    table[h].mutex.unlock();  
}
```

```
h1 = hash(x.key);  
table[h1].mutex.lock();  
x.next = table[h1].next;  
table[h1].next = &x;  
table[h1].mutex.unlock();
```

Thread 1

```
h2 = hash(y.key);  
table[h2].mutex.lock();  
y.next = table[h2].next;  
table[h2].next = &y;  
table[h2].mutex.unlock();
```

Thread 2

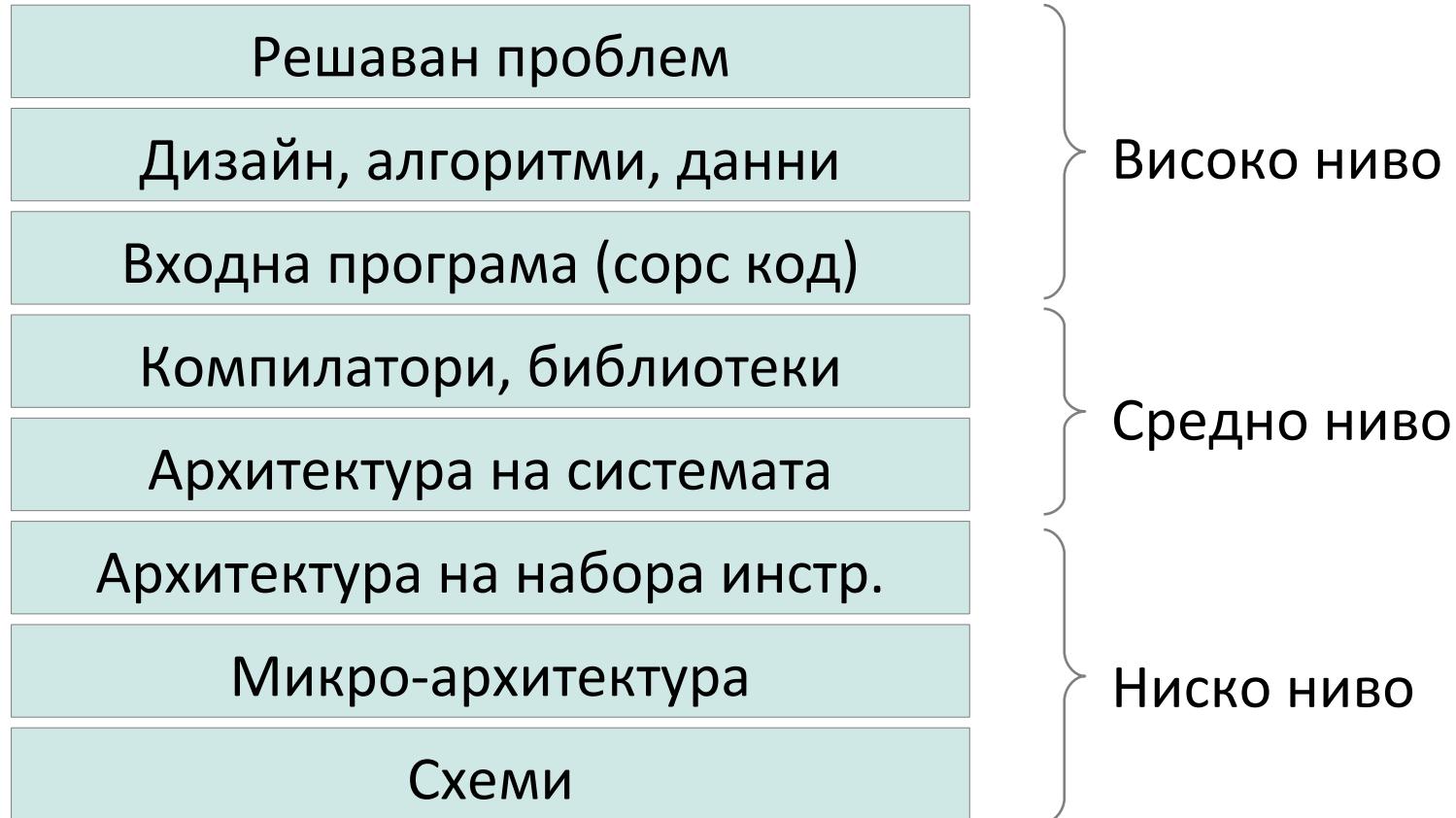
Така е по-добре, защото няма заключване при $h1 \neq h2$



Дизайн на Паралелни програми

- ❖ Анализ на проблема:
 - ❖ Може ли проблема да се разпаралели?
 - ❖ Има ли даннови зависимости?
 - ❖ Идентификация на „тесните“ места.
- ❖ Реализация:
 - ❖ Как да се разпределят данните?
 - ❖ Как да се разпределят инструкциите?
 - ❖ Каква ще е комуникацията?
 - ❖ Необходима ли е синхронизация (защита)?
 - ❖ Можем ли да използваме Fork-Join, Map-and-Reduce или други подобни известни подходи?
- ❖ Тестване – работоспособност, ефективност, скалируемост...;

Обща схема – нива на абстрактност



Проект

Курсов проект

Получавате неефективно работещ проект, който не използва техники от паралелното програмиране.

Вашата задача е:

- ❖ Паралелизация:
 - ❖ Преработвате проекта;
 - ❖ Прилагате техники от ПП;
 - ❖ Анализирате и подобрявате производителността;
- ❖ Оптимизация:
 - ❖ Възползвате се от неефективността на алгоритмите;
 - ❖ Възползвате се от ресурсите на наличната машина(и).
- ❖ Документация.

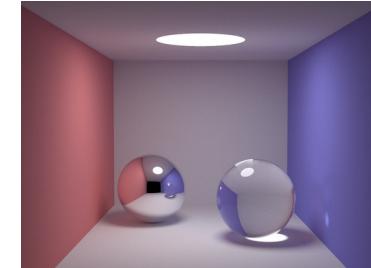
Курсов проект

Имате избор от две възможни теми^{*}:

- ❖ Умножение на две много големи квадратни матрици;

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1N} \\ b_{21} & b_{22} & \cdots & b_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N2} & \cdots & b_{NN} \end{pmatrix} = ?$$

- ❖ Визуализация на 3D сцени с помощта на Ray-tracing;



Няма единствен „верен отговор“:

- ❖ Имате много свобода (и малко напътствия);
- ❖ Обаче има лесен начин да се види кой е най-доброят;

* Виж примерните проекти в сайта с помощни материали

Политика за работа (можете да)

- ❖ Използвате произволен език за програмиране;
- ❖ Използвате произволна среда за разработка;
- ❖ Използвате произволна операционната система;
- ❖ Използвате произволни методи и библиотеки за паралелно програмиране и оптимизация;
- ❖ Използвате примерните проекти (виж помощните материали), като основа за създаване на собствен проект или като пример за не паралелна реализация на съответната задача;
- ❖ Всеки работи самостоятелно;
- ❖ Допустимо е да обсъждате проблема, изискванията и материалите с всеки;

Политика за работа (не можете да)

- ❖ Не може да използвате проекти (или части от тях) създадени от други лица за целите на този курс, включително създадени в предишни учебни години;
- ❖ Не може да използвате проекти (или части от тях) създадени от други лица за сходни цели;
- ❖ Не може да използвате код, който не сте реализирали собственоръчно (с изключение на код от примерните проекти);

Предаване на проекта

Всеки трябва да работи по проекта си по време на семестъра и
да предаде проекта си
(сурс код + документация описваща направените промени)
в хранилището си в SVN подпапка **/trunk/PPProject**
поне два дни преди изпита.

Успех!

Въпроси?

apenev@uni-plovdiv.bg



Паралелно Програмиране

Съвременни паралелни архитектури.

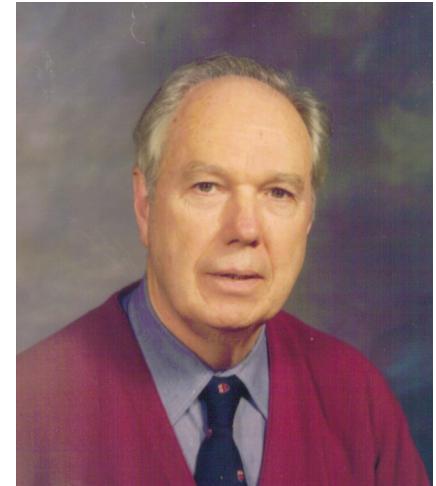
Класификация на Флин.

CPU, GPGPU, HSA, HPC и др.

доц. д-р Александър Пенев

Класификация на Флинн /Flynn/

Класификация на Флин /Flynn/ (1966)



SISD

Старите Mainframe,
класическите PC и др.

SIMD

Повечето съвременни
PC, GPU и др.

MISD

Използва
за системи защитени
от повреди,
С.тмр computer и др.

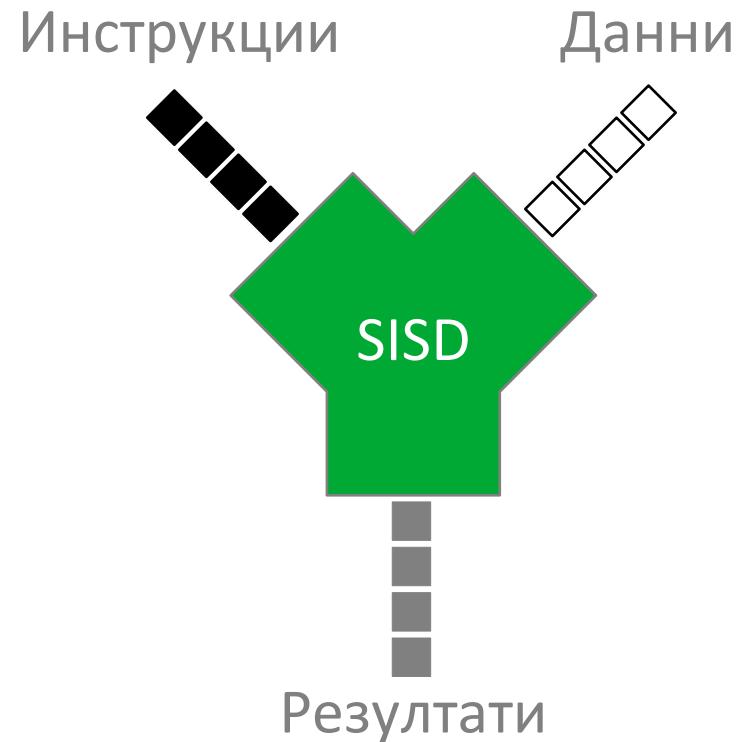
MIMD

Много-ядрените
суперскаларни проц.,
разпределените
системи и др.

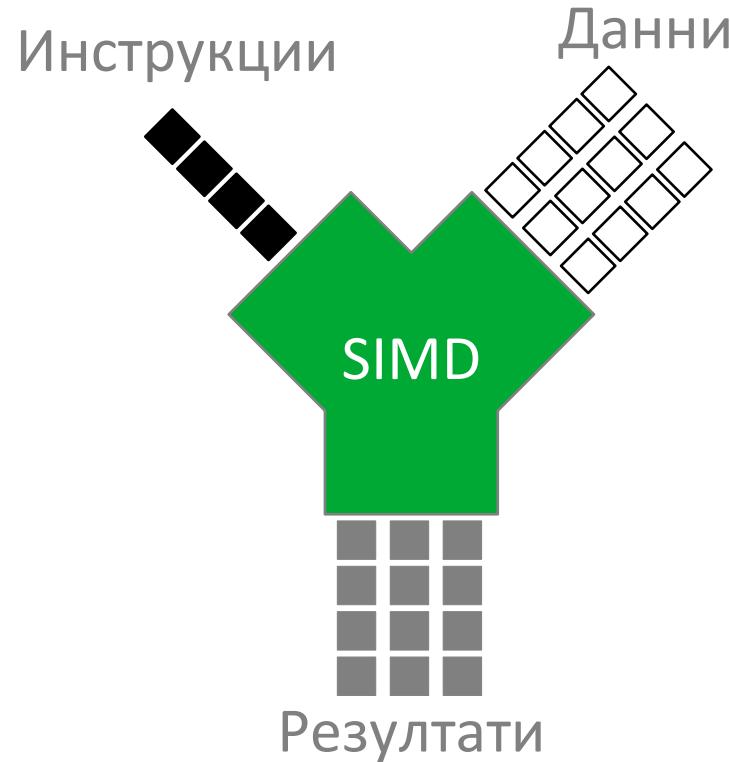
Класификация на Флин /Flynn/

- ❖ SISD – един поток инструкции над едни поток данни;
- ❖ SIMD – един поток инструкции над много потоци данни;
- ❖ MISD – много потоци инструкции над едни поток данни;
- ❖ MIMD – много потоци инструкции над много потоци данни;
 - ❖ SPMD – една програма, много потоци данни;
 - ❖ MPMD – много програми, много потоци данни;

SISD



SIMD



MISD

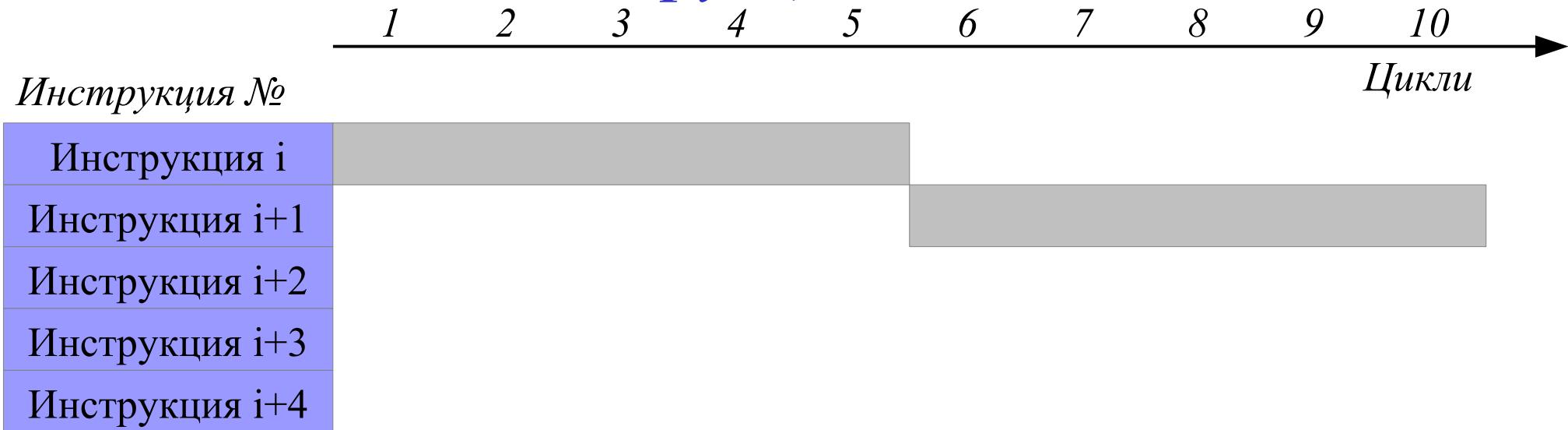


MIMD



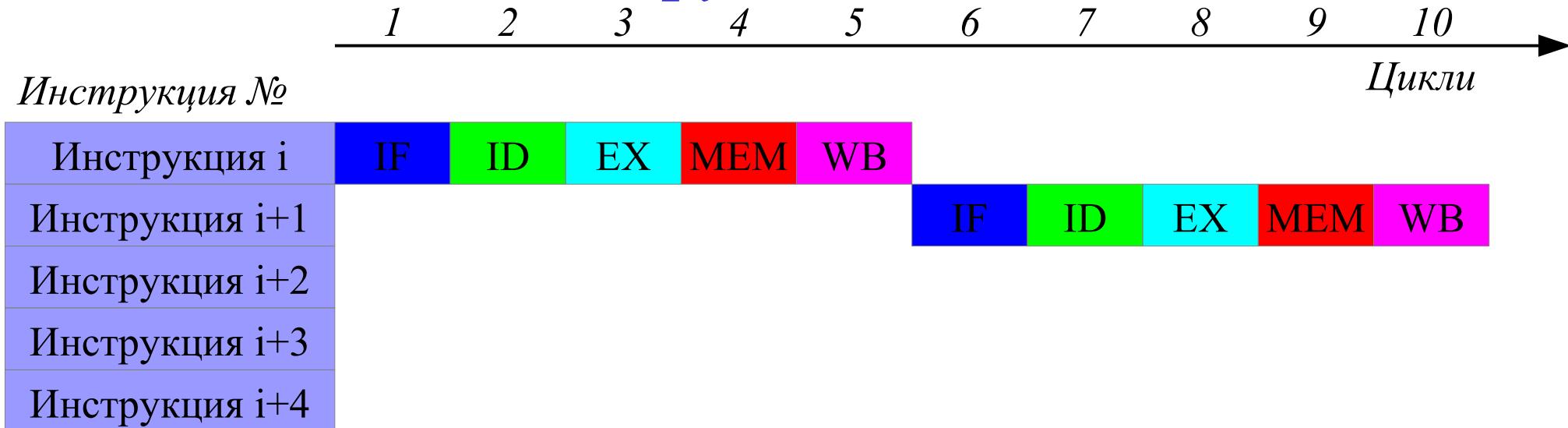
Класически Процесори

Изпълнение на инструкции



Последователно изпълнение на инструкциите.

Изпълнение на инструкции

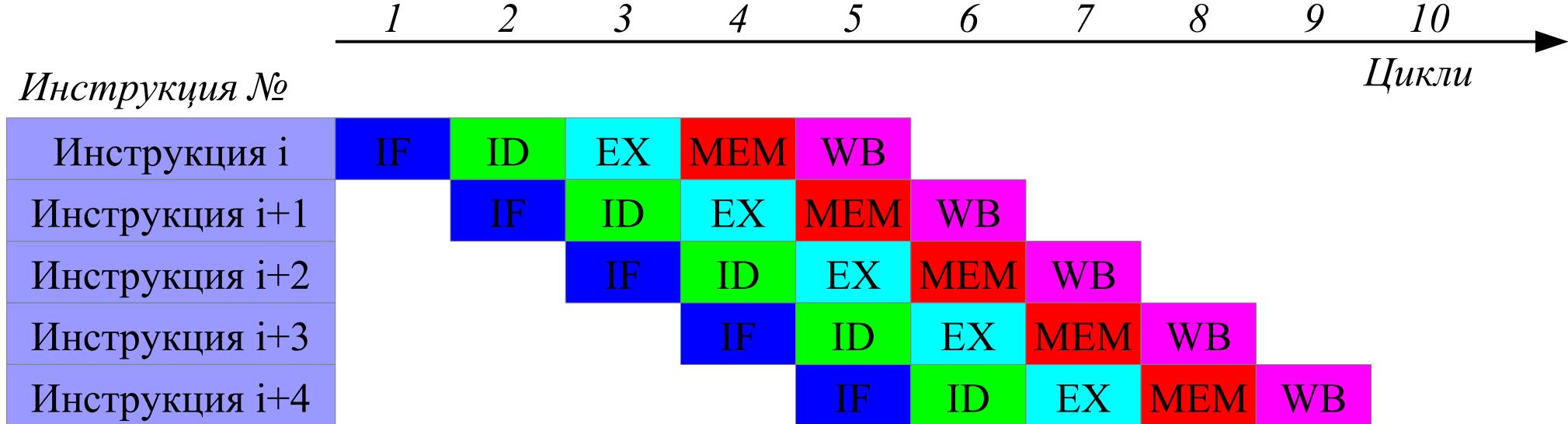


Но самите инструкции обикновено се състоят от отделни под етапи.

- ❖ IF – Instruction Fetch
- ❖ ID – Instruction Decode
- ❖ EX – Execution
- ❖ MEM – Memory Access
- ❖ WB – Write Back

Скалярни/Pipelined Процесори

Изпълнение на инструкции (Pipelined/Scalar)



Характерно е и за
класическите RISC
архитектури.

- ❖ IF – Instruction Fetch
- ❖ ID – Instruction Decode
- ❖ EX – Execution
- ❖ MEM – Memory Access
- ❖ WB – Write Back

Ограничения при конвейерното изпълнение

Опасностите (hazards) не позволяват следващата инструкция да бъде изпълнена по време на определения ѝ цикъл. Те се делят на:

- ❖ **Даннови опасности (data hazards)**

Инструкция зависи от резултата на друга инструкция, която е още в конвейера

- ❖ **Контролни опасности (control hazards)**

Причинени са от забавянето между извлечането на инструкции и решението за промени в потока инструкции (control flow) – условни, безусловни и индиректни преходи, подпрограми

- ❖ **Структурни опасности (structural hazards)**

Опити за използване на един и същ хардуер за две различни неща едновременно

Даннови опасности: без зависимост

Инструкция 3 не зависи от данните, получени от инструкция 2.

В двете инструкции се чете само от данните получени в инструкция 1.

1. $A = 3$
2. $B = A$
3. $C = A$

Ако две инструкции не са взаимнозависими от данните си те обикновено могат да бъдат:

- ❖ Изпълнени едновременно;
- ❖ Напълно да се застъпват в конвейера;
- ❖ Да се изпълняват непоследователно (out-of-order);

При настъпване на такава ситуация в конвейера се казва, че имаме четене след четене (RAR), което не е риск



Даннови опасности: истинска зависимост

Инструкция 3 зависи от
данные, получены от
предходната инструкция 2

1. $A = 3$
2. $B = A$
3. $C = B$

Ако две инструкции са взаимозависими от данните си те не могат да бъдат:

- ❖ Изпълнени едновременно;
- ❖ Напълно да се застъпват в конвейера;
- ❖ Да се изпълняват непоследователно (out-of-order);

При настъпване на такава ситуация в конвейера се казва, че има риск от
чтение след запис (RAW hazard)



Даннови опасности: анти зависимост

Инструкция 3 зависи от
инструкция 2 (инстр. 2 трябва да
се изпълни преди инстр. 3)

1. $B = 3$
2. $A = B + 1$
3. $B = 7$

Нарича се още зависимост от имената

Когато две инструкции използват един и същи регистър или адрес в паметта, но няма поток от данни между тези две инструкции. Има две версии на този тип зависимост

При настъпване на такава ситуация в конвейера се казва, че има рисък от запис след четене (WAR hazard)



Даннови опасности: изходна зависимост

Инструкция 3 зависи от
инструкция 1 (инстр. 1 трябва да
се изпълни преди инстр. 3)

1. **A** = 2 * X
2. **B** = A / 3
3. **A** = 9 * Y

Нарича се още зависимост от имената

Когато две инструкции използват един и същи регистър или адрес в паметта, но няма поток от данни между тези две инструкции. Има две версии на този тип зависимост

При настъпване на такава ситуация в конвейера се казва, че има рисък от запис след запис (WAW hazard)



Зависимост “от имената”

Зависимостта от имената може да се избегне като се смени името в инструкцията, така че да няма конфликт:

- ❖ Софтуерно преименуване (на регистри)

Прави се от компилатора

- ❖ Хардуерно преименуване (на регистри)

Прави се от процесора. Недостатък: по време на изпълнение

```
1. B = 3  
2. A = B + 1  
3. B = 7
```

```
1. B = 3  
2. B2 = B  
3. A = B2 + 1  
4. B = 7
```

WAR опасност

```
1. A = 2 * X  
2. B = A / 3  
3. A = 9 * Y
```

```
1. A2 = 2 * X  
2. B = A2 / 3  
3. A = 9 * Y
```

WAW опасност

Контролни опасности

Всяка инструкция е зависима от някакво множество преходи. Тези зависимости трябва да се спазват, за да може да се запази семантиката на програмата

```
if c1 {  
    I1;  
}  
if c2 {  
    I2;  
}
```

I1 зависи от условието c1, а I2 зависи от условието c2, но не и от c1.

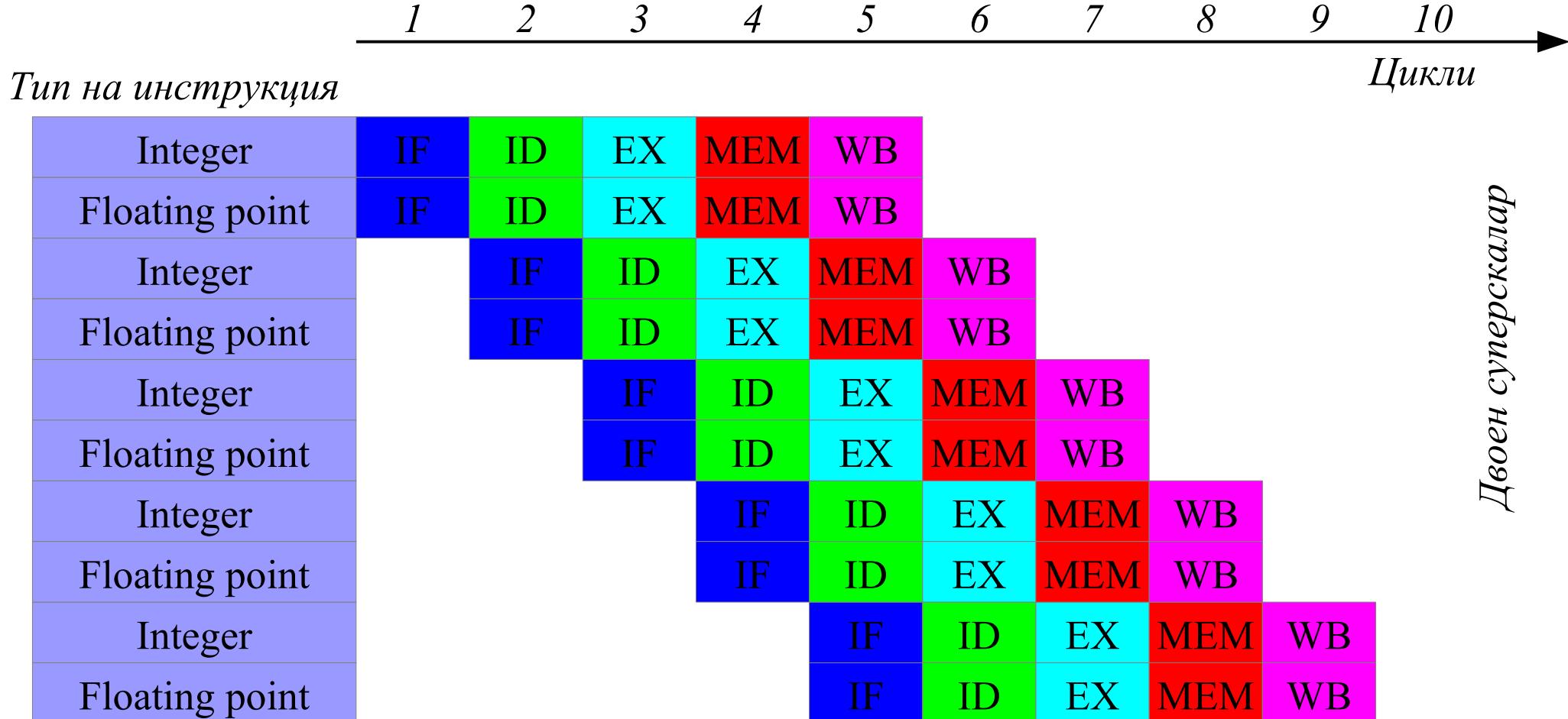
Контролни опасности

Избягването на тези конфликти е трудно. Единственият начин е да не се спазва последователността от изпълнение. С други думи изпълнение на инструкции, които не би трябвало да се изпълняват и следователно водят до нарушаване на контролните зависимости. Това е приложимо само когато може да се запази семантиката на програмата.

Този вид изпълнение се нарича **спекулативно изпълнение**.

Суперскаларни Процесори

Суперскалярно изпълнение



Двоен суперскаляр

Даннови опасности при ILP

- ❖ Локализация на Instruction level parallelism

Много инструкции да се изпълняват паралелно

- ❖ Хардуерът/Софтуерът трябва да запази редът на изпълнение на програмите

Трябва да се запази редът на изпълнение на инструкциите и да изглежда така че те се изпълняват от последователно и следват сорс кода

- ❖ Важност на данновите зависимости:

- ❖ Отбелязва възможни опасности, а не задължителни проблеми
- ❖ Определя редът, в който резултатите трябва да се калкулират
- ❖ Ограничава отгоре колко паралелизъм може да се използва

Крайна цел: използване на паралелизма като се запази последователното изпълнение само когато то има негативен ефект върху изпълнението на програмата



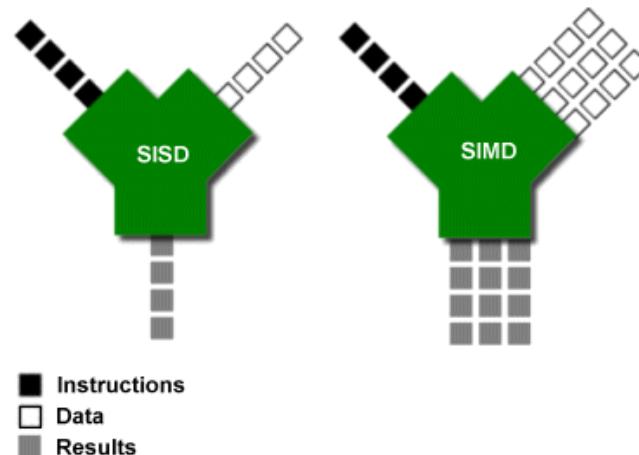
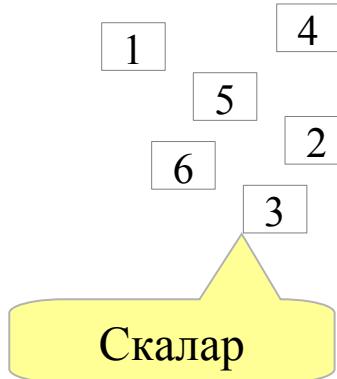
SMID инструкции

- ❖ SIMD (Single Instruction Multiple Data)

Техниката се използва, за да се постигне паралелизъм на ниво данни като при векторните процесори.

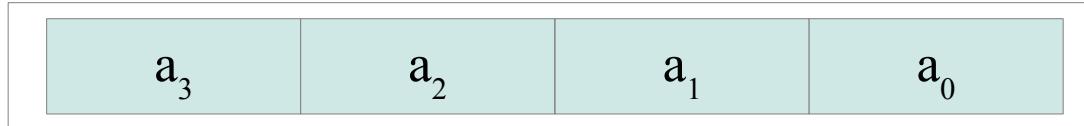
- ❖ Една от най-новите реализации на SIMD от Интел се нарича AVX-512. Преди нея се появяват: MMX, SSE, ..., AVX, AVX2 и др.

SIMD инструкции

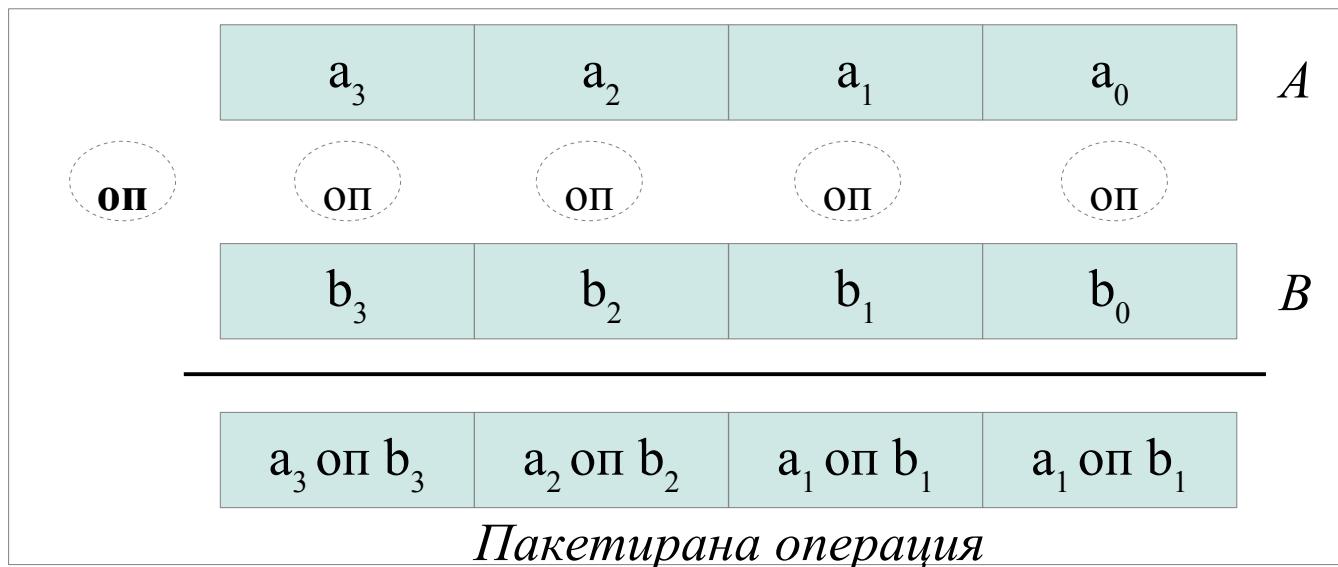


SIMD инструкции

- ❖ Пакетиран тип данни – представлява вектор от няколко скалара
Намират се в отделени специализирани регистри



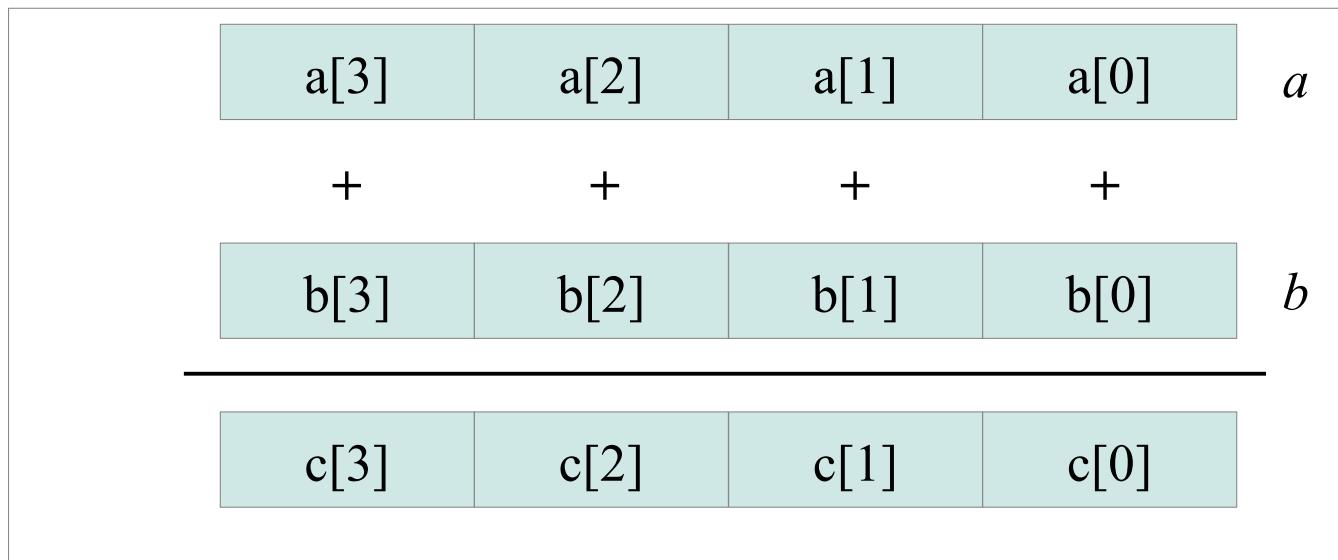
- ❖ SIMD



SIMD инструкции – пример “събиране”

```
for(i=0; i< MAX; ++i)  
    c[i] = a[i] + b[i];
```

❖ SIMD



Intel Core Микроарх. Суперскалярно изпълн.

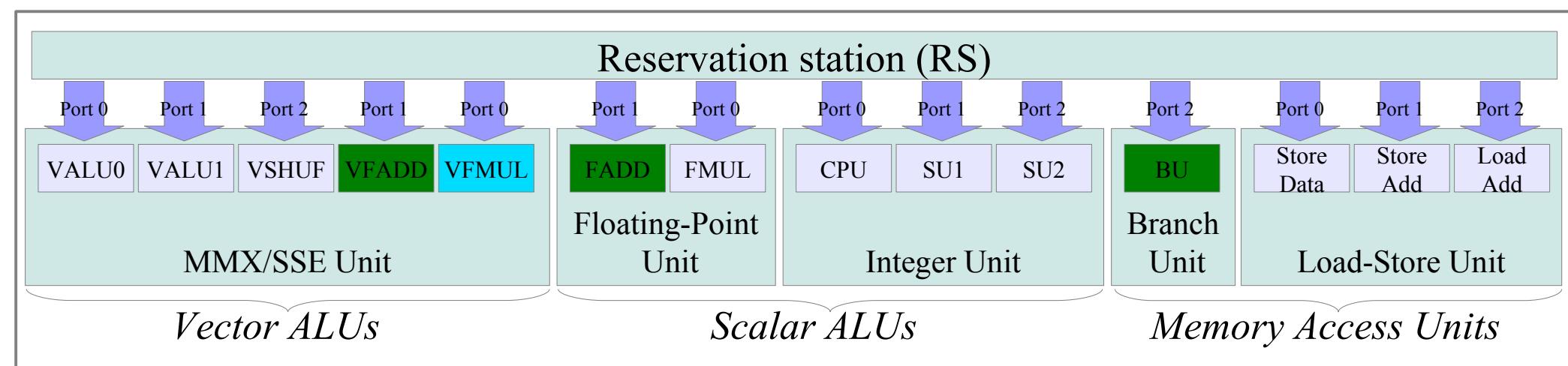
- ❖ 3 Load/Store

- ❖ 1 Branch

- ❖ 3 Integer

- ❖ 2 Floating point

- ❖ 5 SSE



Спекулативно изпълнение

Различни механизми за предсказване

Постига се по-добра ILP като се преодоляват контролните зависимости чрез спекулиране на резултата от преходите и изпълнява програмата сякаш предсказанията са верни

❖ Предсказване на преходи

Предвиждане на условни, безусловни и изчислими преходи, както и на извикване и връщане от подпрограми. В повечето случаи с много голяма вероятност за успех евъзможно да се предскаже посоката на прехода, както и много често адреса на прехода

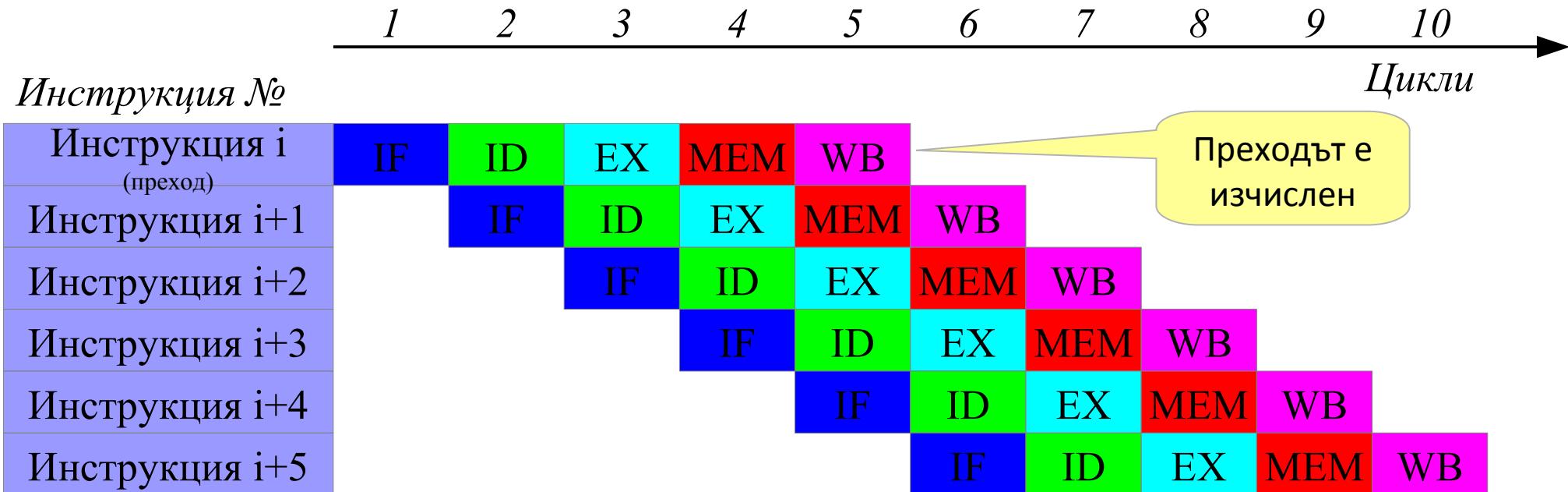
❖ Предсказване на стойности

Предвиждане на стойностите на регистри и други

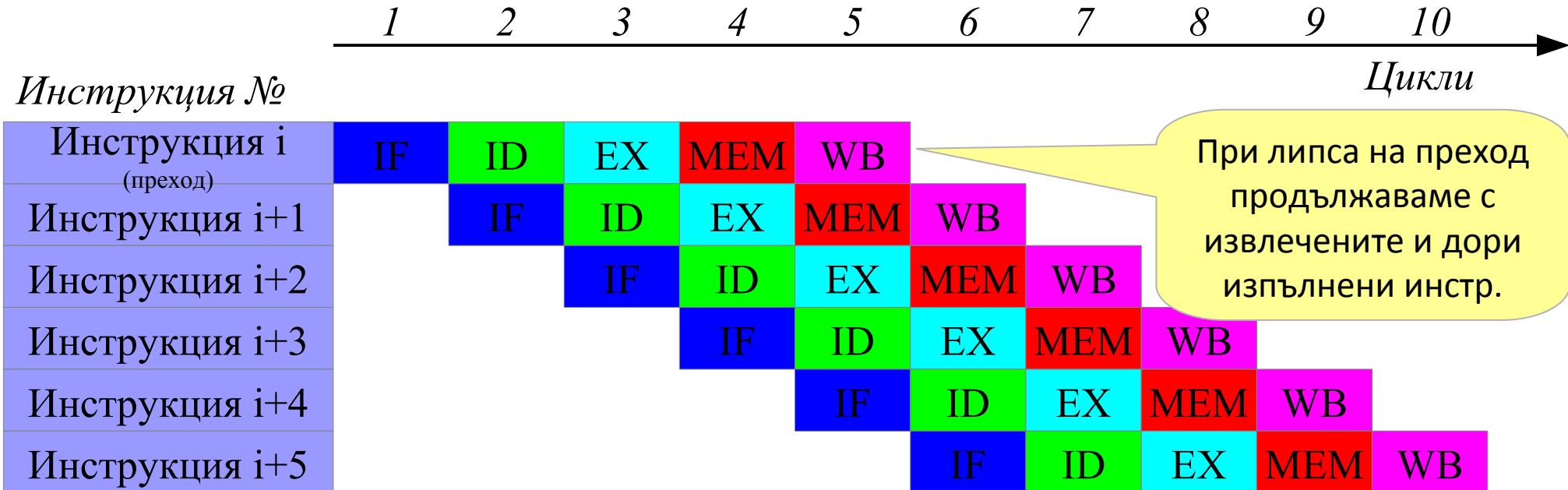
❖ Предварително извлечане (Prefetching)

Предвиждане на начинът на достъп до паметта. Повечето достъп до паметта не е напълно хаотичен. Различните инструкции слдват накакви очаквани шаблони, например достъпа до стека е предишните/следващите елементи спрямо SP регистъра (Push записва в предишните, Pop извлича от следващите). Това предвиждане дава възможност за предварително извлечане на данните от очакваните адреси наведнъж (предварително)

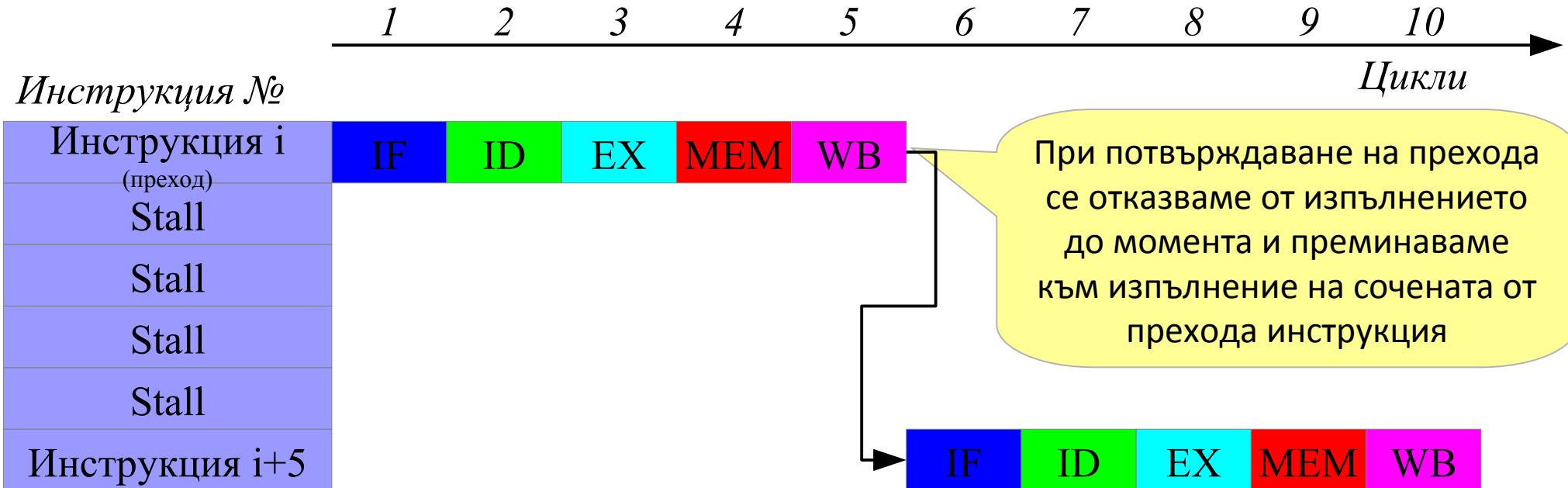
Предсказване на переходи и спекулативно изпълн.



Предсказване на преходи и спекулативно изпълн.



Предсказване на преходи и спекулативно изпълн.



Предсказване на преходи и спекулативно изпълн.

Механизмът за предсказване трябва да определи кое от двете разклонения на условието ще се изпълни

- ❖ Съвременните предсказващи системи са 99+% точни

Дори могат да предсказват адресите от индиректните/изчислени преходи.

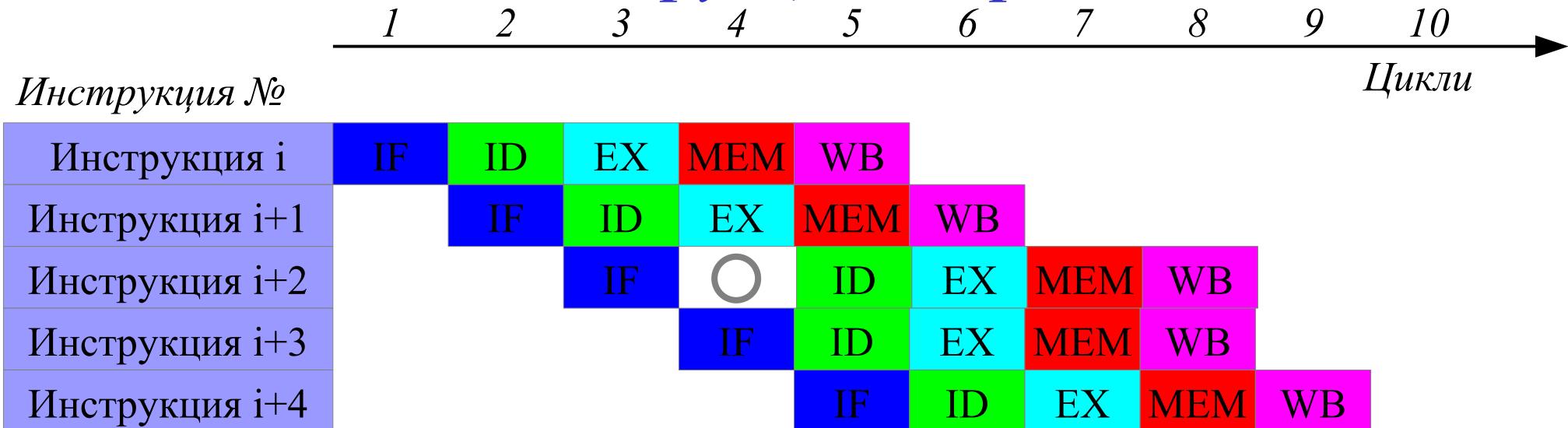
Понеже грешно предсказаните преходи са много малък процент, то печалбата от правилно предсказаните преходи е много по-голяма от загубата при грешно предсказание

Извличат се и се изпълняват спекулативно предсказани адреси

- ❖ Няма “мехурчета” в конвейера

Когато преходът най-накрая е оценен и адресът е ясен тогава спекулативното изпълнение се потвърждава или отхвърля

Изпълнение на инструкции – при “опасност”



При невъзможност да се изпълни даден етап от инструкцията, се отлага изпълнението му чрез вмъкване на “мехурче” в конвейера т.е. NOP операция.

Векторни Процесори

Векторни процесори

В суперкомпютъра Cray-1
за първи път успешно се
реализира концепцията
за векторни процесори.



Векторни процесори

- ❖ Основната “печалба” не е от намаление брой декодирания на инструкции, а идва от скриване на латентността на достъпа до паметта поради извличане на големи обеми данни от последователни адреси в паметта;
- ❖ Изпълняват SIMD векторни инструкции върху вектори с произволна дължина (или по-често с вектори с фиксирана дължина);
- ❖ Някои реализации изпълняват инструкции от вида Памет-Памет, но по-успешните подходи са с използване на големи векторни регистри;

Dataflow Архитектура

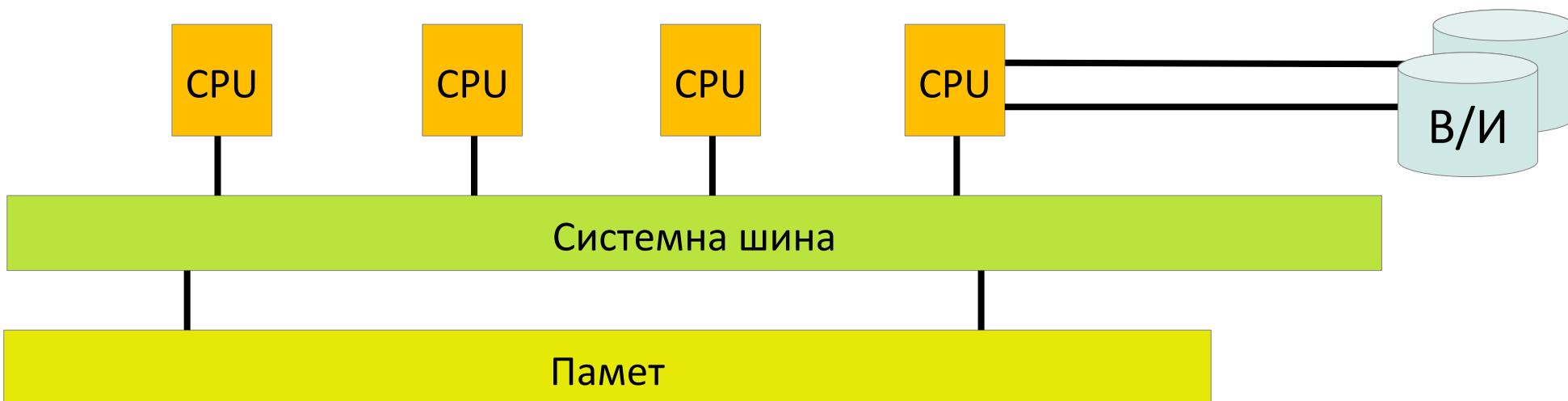
Dataflow архитектура

- ❖ В класическите dataflow архитектури последователността на изпълнението се базира на вида на параметрите (данныте), вместо на последователността на контрола. Това изисква ефективна реализация на асоциативна памет и води до недетерминираност на изпълнението;
- ❖ За сега няма успешна реализация на хардуер базиран на тази архитектура, но има много примери на специализиран хардуер, като например Digital Signal Processing, Network routing, Graphics Processing и др.;
- ❖ Поради силно паралелната природа на dataflow подхода той се прилага в много dataflow базирани езици за програмиране;

Мултипроцесорни Системи (Symmetric/Asymmetric)

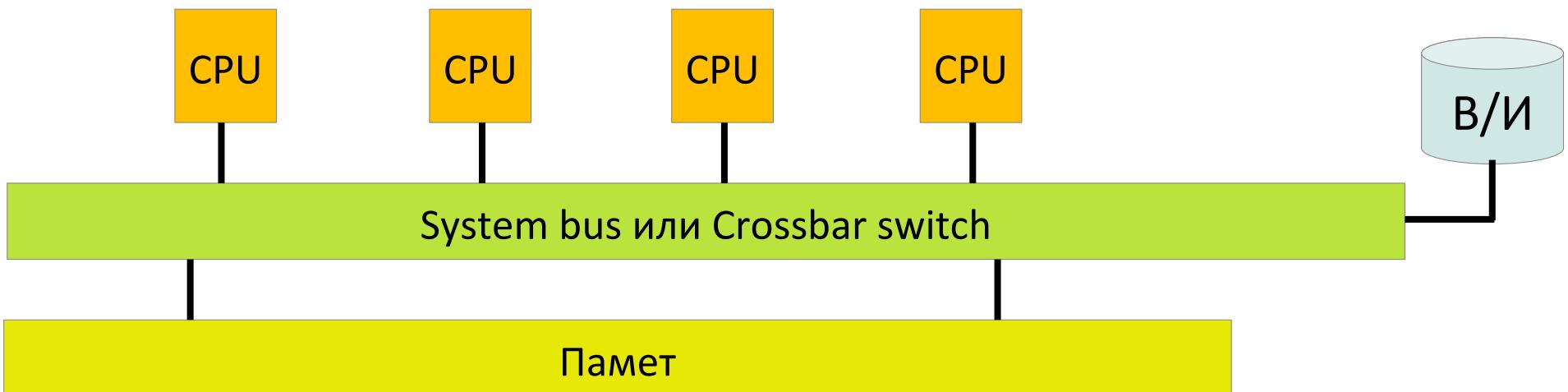
Мултипроцесорни системи

- ❖ Асиметрични – някои от процесорите се различават по предназначението си в системата. Например ОС се изпълнява само на някои процесори и т.н.;



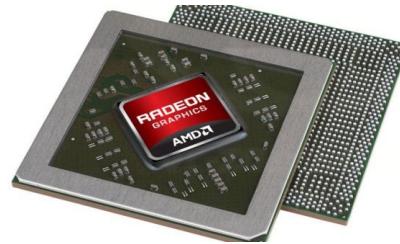
Мултипроцесорни системи

- ❖ Симетрични – всички процесори са “равноправни” в системата;

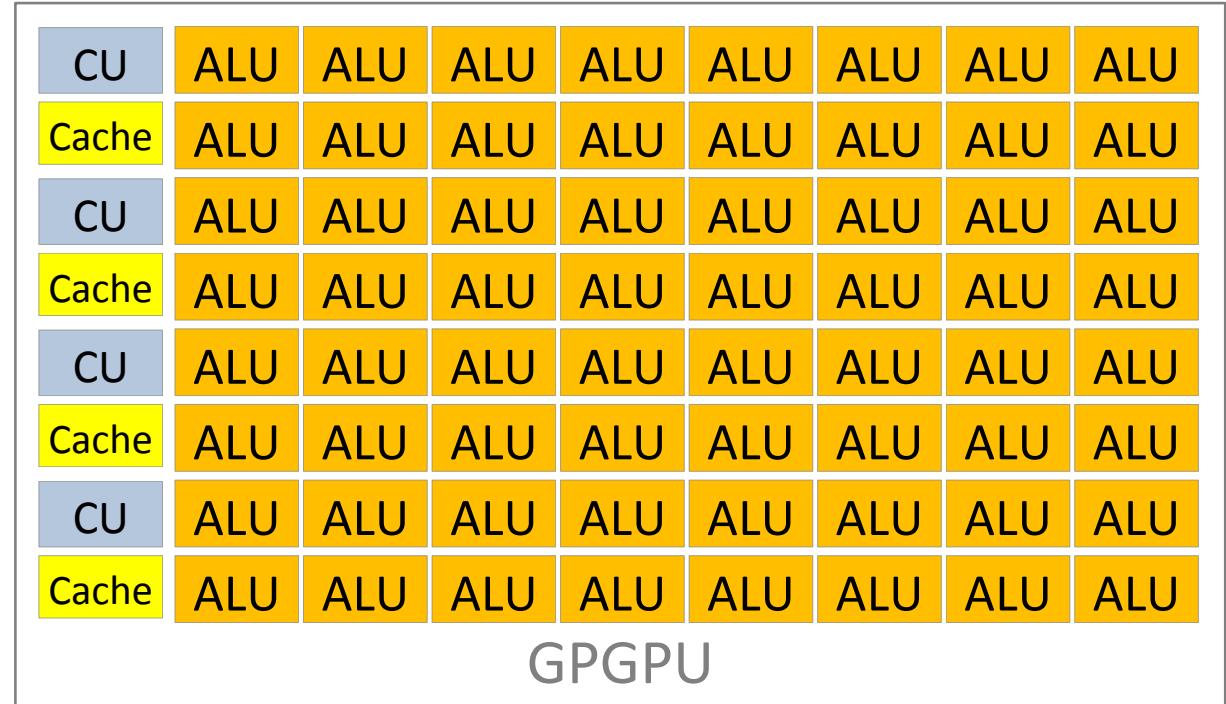
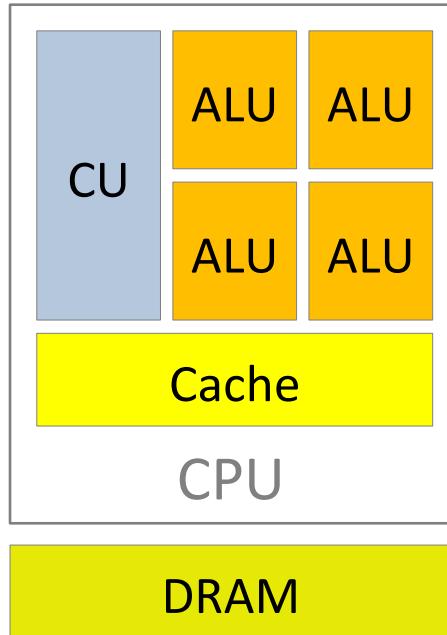


GPU, GPGPU

GPU, GPGPU, ...



CPU vs. GPGPU



CPU vs. GPGPU

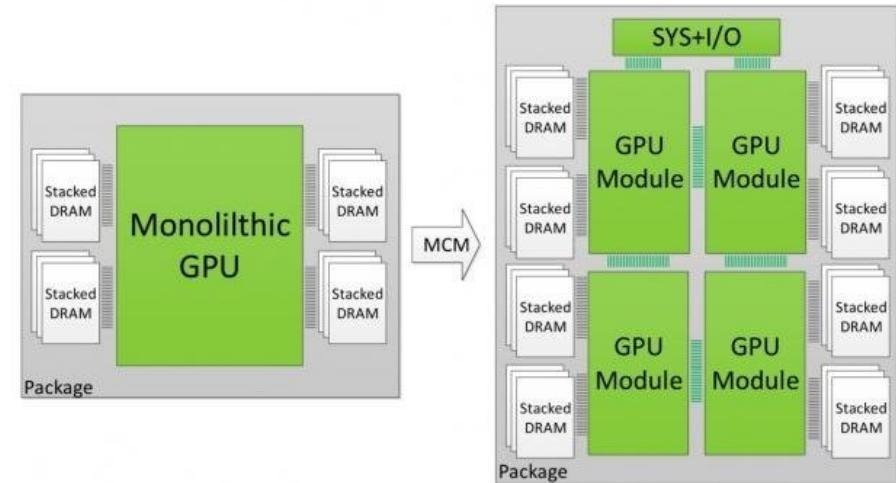
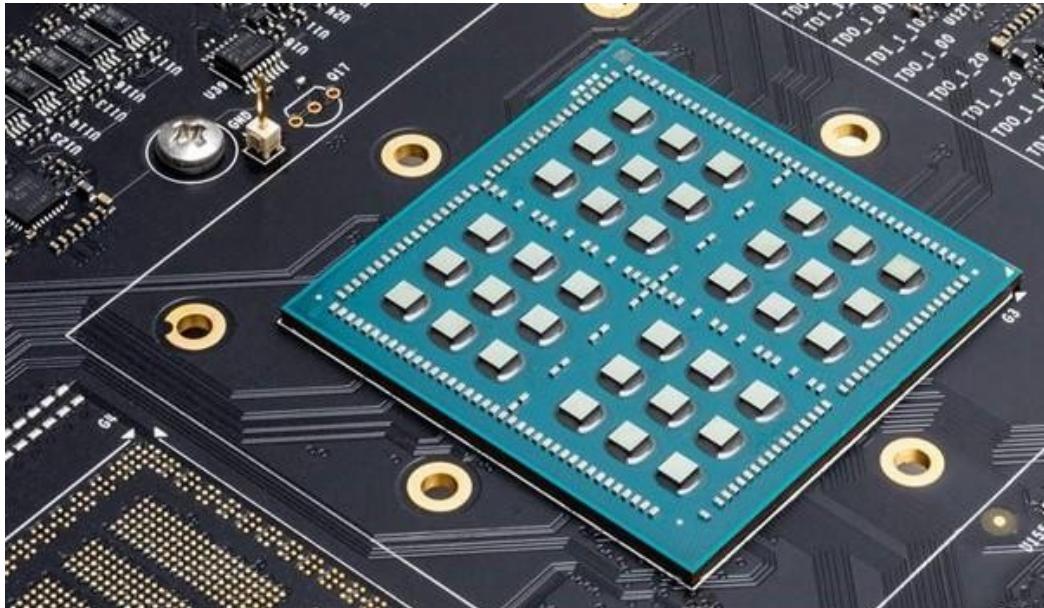
CPU

- ❖ Има нужда от много памет;
- ❖ Скороста е по-ниска;
- ❖ Съдържа малък брой много мощни ядра;
- ❖ Подходящ е за последователно изпълнение;
- ❖ Ниска латентност;

GPGPU

- ❖ Използва по-малко памет;
- ❖ Много висока скорост;
- ❖ Състои се от много на брой “леки” ядра;
- ❖ Създаден е за силно паралелно/стрийм изпълн.;
- ❖ Висока пропускливост;

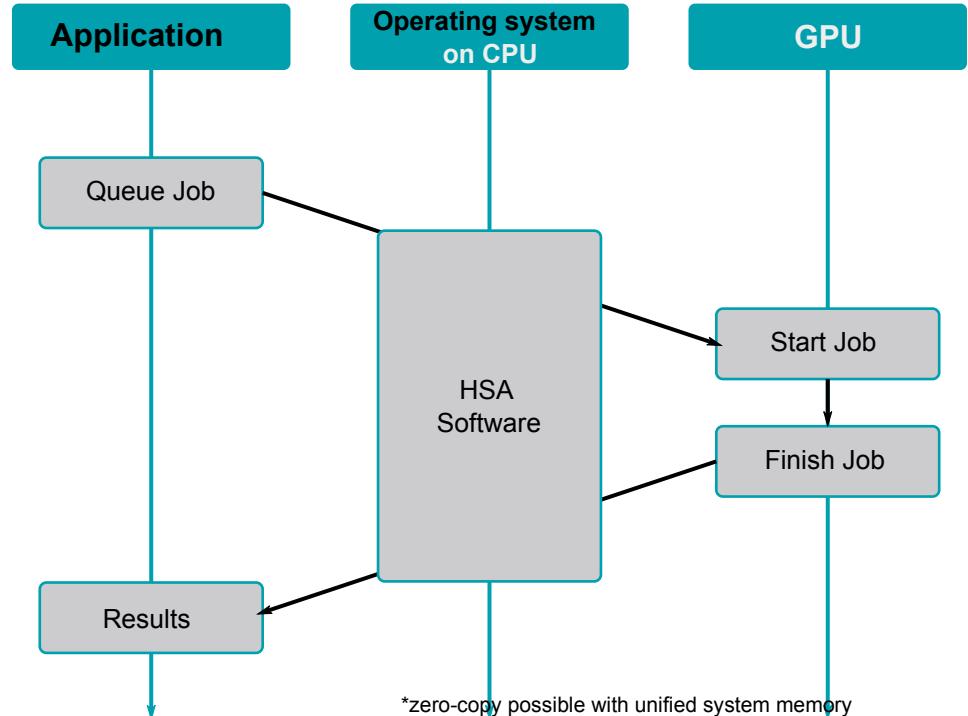
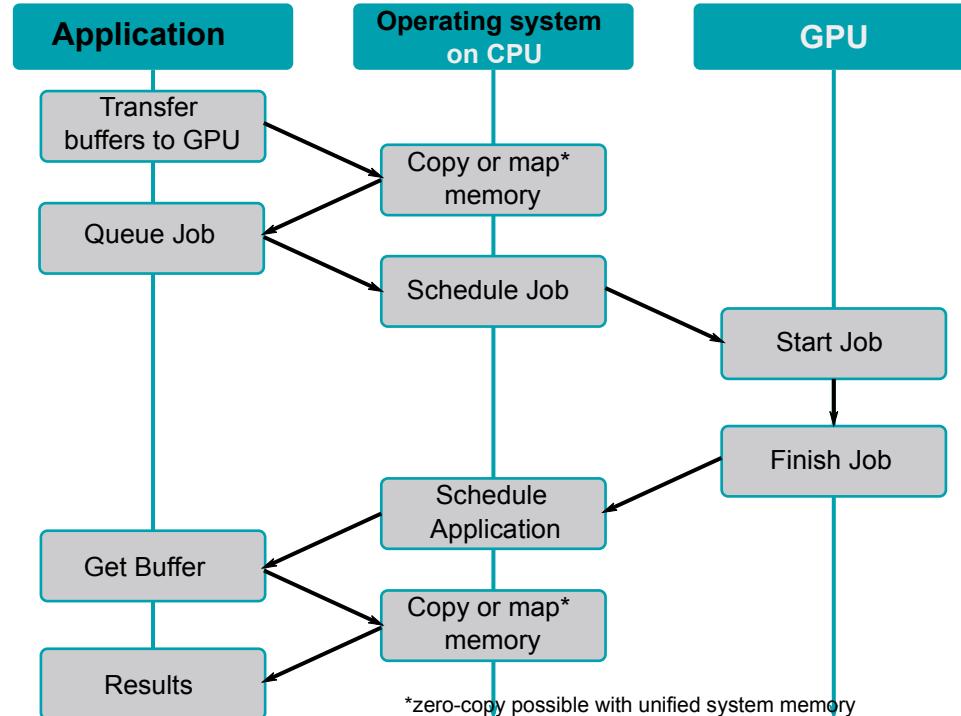
NVIDIA Next Gen-GPU Hopper



Новата архитектура на
nVidia Hopper

Хетерогенна Архитектура (HSA)

Heterogeneous System Architecture (HSA)



Високо Производителни Компютри (HPC)

Високо Производителни Компютри (HPC)

IBM Summit

суперкомпютър с производителност 200 petaFLOPS

Процесори:

POWER9 – 9216 бр.

Tesla V100 – 27648 бр.

Памет:

600GB RAM + 800GB NvRAM

Връзка:

InfiniBand EDR



Високо Производителни Компютри (HPC)

IBM Blue Gene/P
суперкомпютър с
164000 процесорни
ядра



Въпроси?

apenev@uni-plovdiv.bg



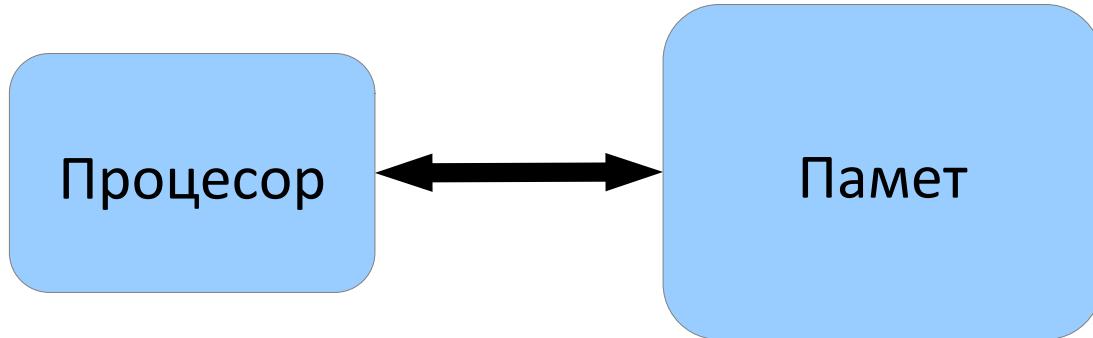
Паралелно Програмиране

Памет. Йерархия на паметта.
Архитектури на паметта на
паралелните компютри

доц. д-р Александър Пенев

Йерархия на Паметта

“Тясно гърло” при комуникацията



Производителността на високо-скоростните паралелни компютри обикновено е ограничена от *пропускателната способност и латентността*.

- ❖ **Латентност**

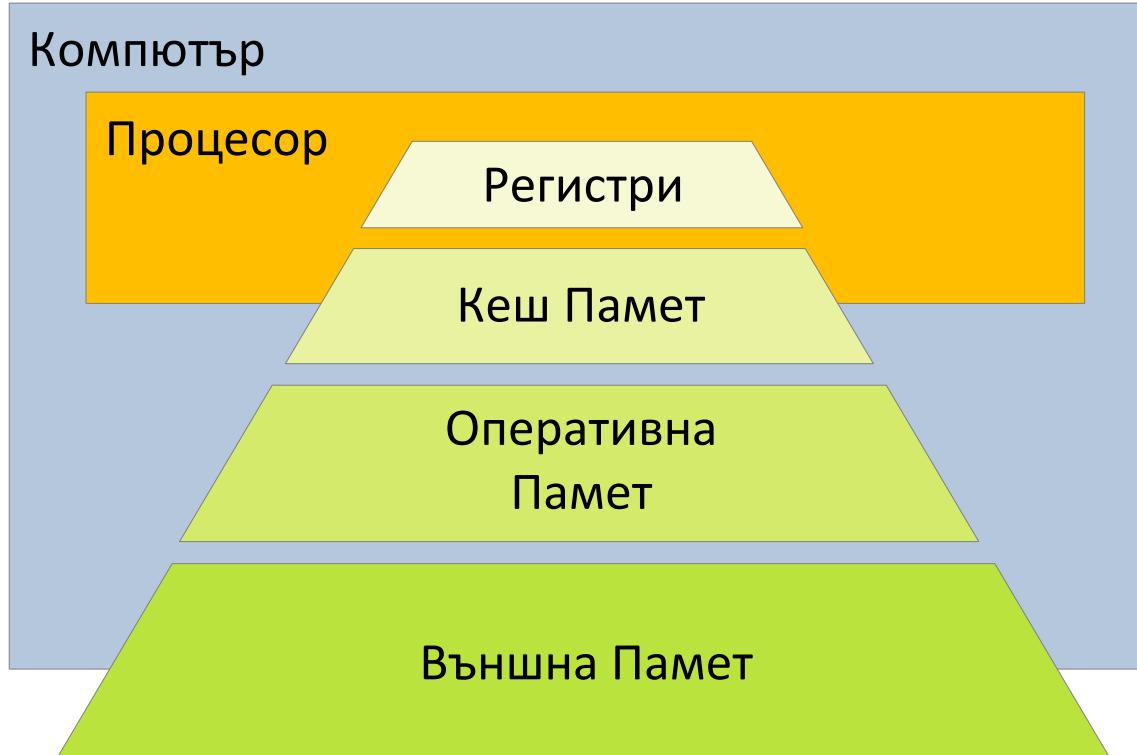
Времето за достъп до паметта >> Времето на цикъл на процесора

- ❖ **Пропускателна способност**

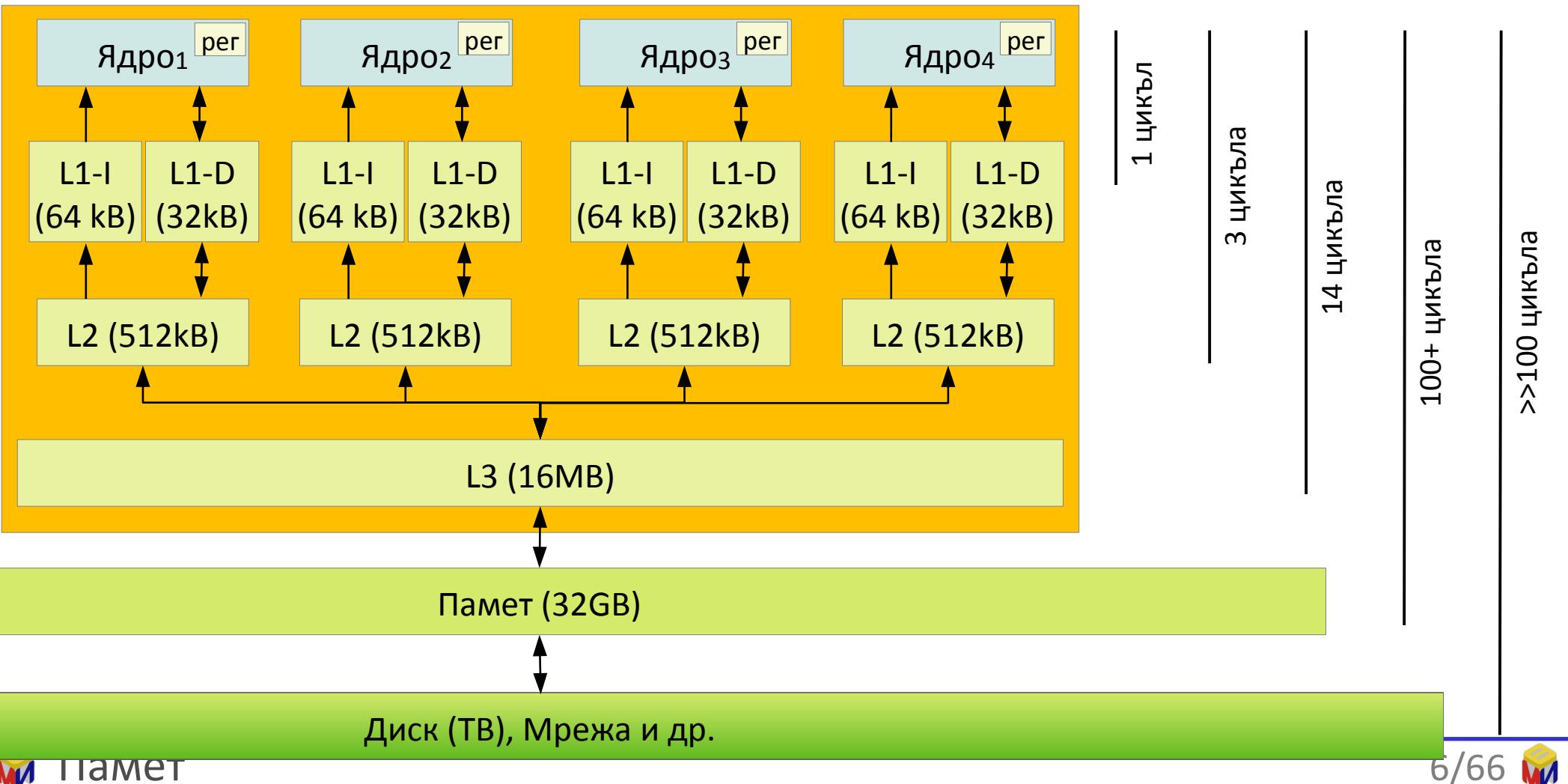
Максималният брой достъпи за единица време

Разбирането на работата на (достъпа до) паметта (и особено кеша) в съвременните (високо производителни и паралелни) системи е много важен фактор за разбирането и програмирането на ефективен паралелен софтуер.

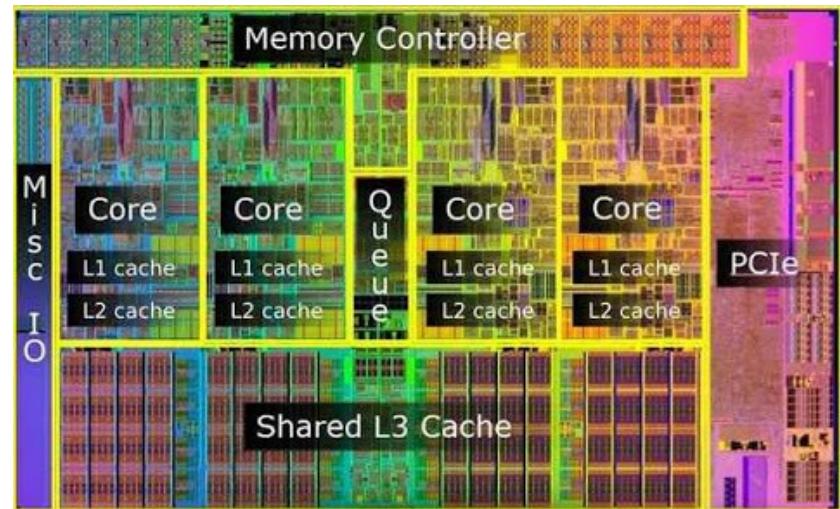
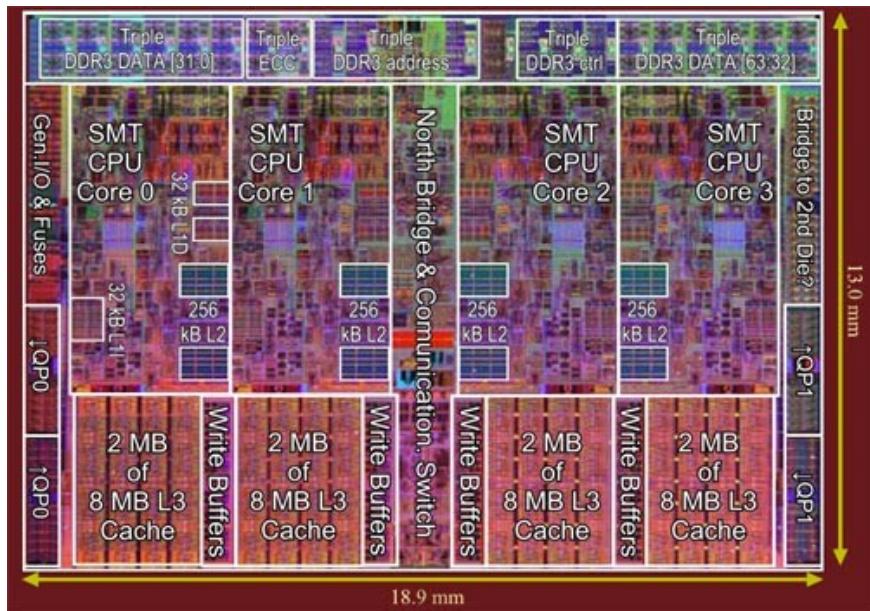
Йерархия на паметта



Йерархия на паметта – Пример



Йерархия на паметта – Пример



Памет в компютърната система

Основния проблем:

- ❖ Малък обем памет – бърз достъп;
- ❖ Голям обем памет –бавен достъп;
- ❖ Как програмата може да използва много памет и да има максимално бърз достъп до нея?

Йерархия на паметта

Задача на кеш система:

- ❖ Да съхранява най-често и най-вероятно използваните стойности в памет с много ниски времена на достъп;
- ❖ Хардуерни евристики определят какво и кога да бъде включено в кеш система;

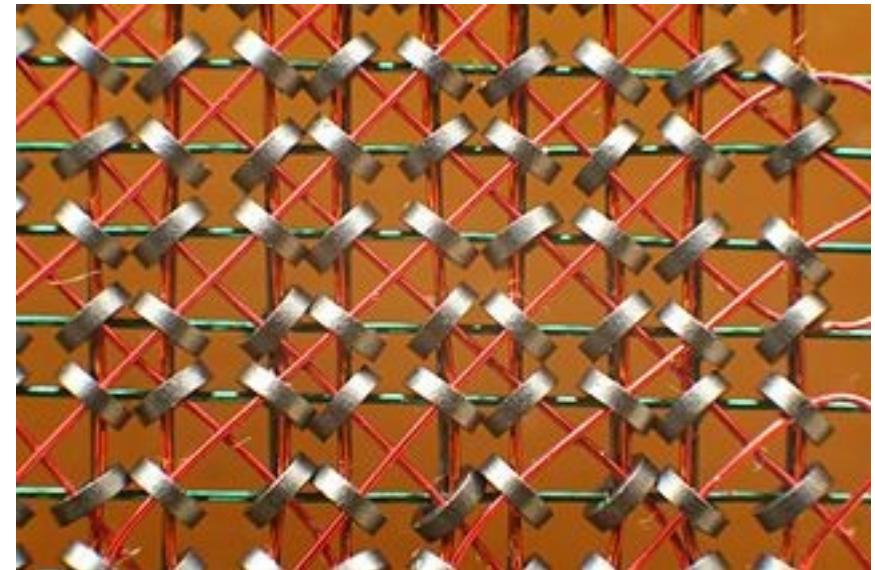
Какво става, ако начинът на достъп в програмата се различава от хардуерната евристика?



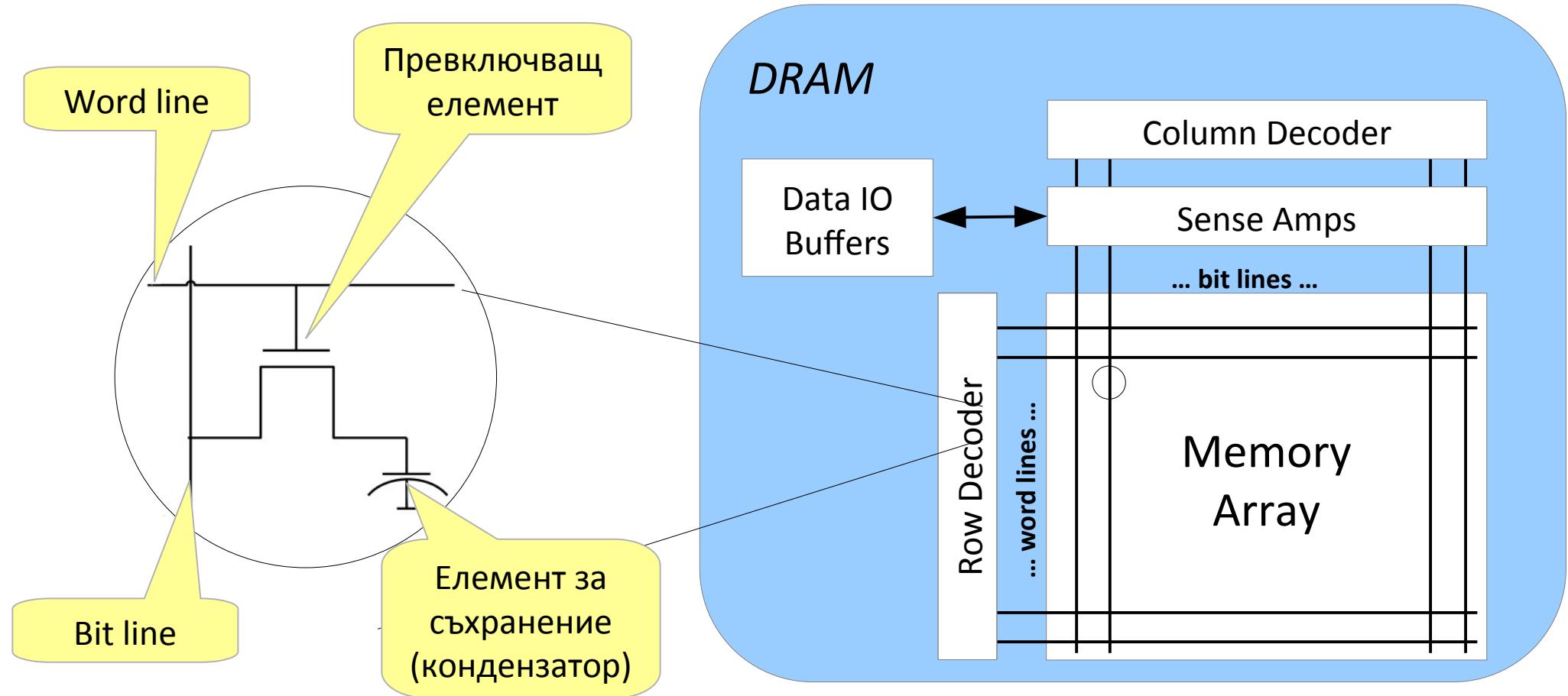
Памет

Core Memory

- ❖ Първата надеждна оперативна памет
- ❖ Предшественик на съвременната памет с произволен достъп (RAM)
- ❖ Битовете съхранени на магнетизирани ядра свързани в двумерна мрежа
- ❖ Използваше се до скоро в совалките на космическата програма на NASA

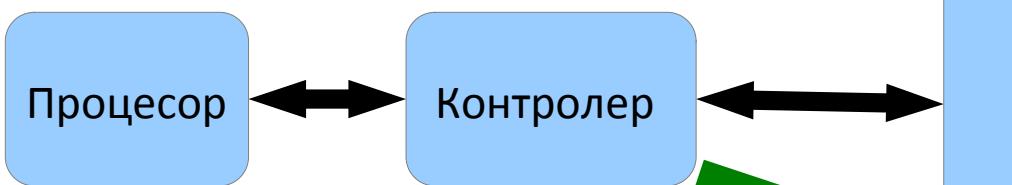


Памет от полупроводници. RAM



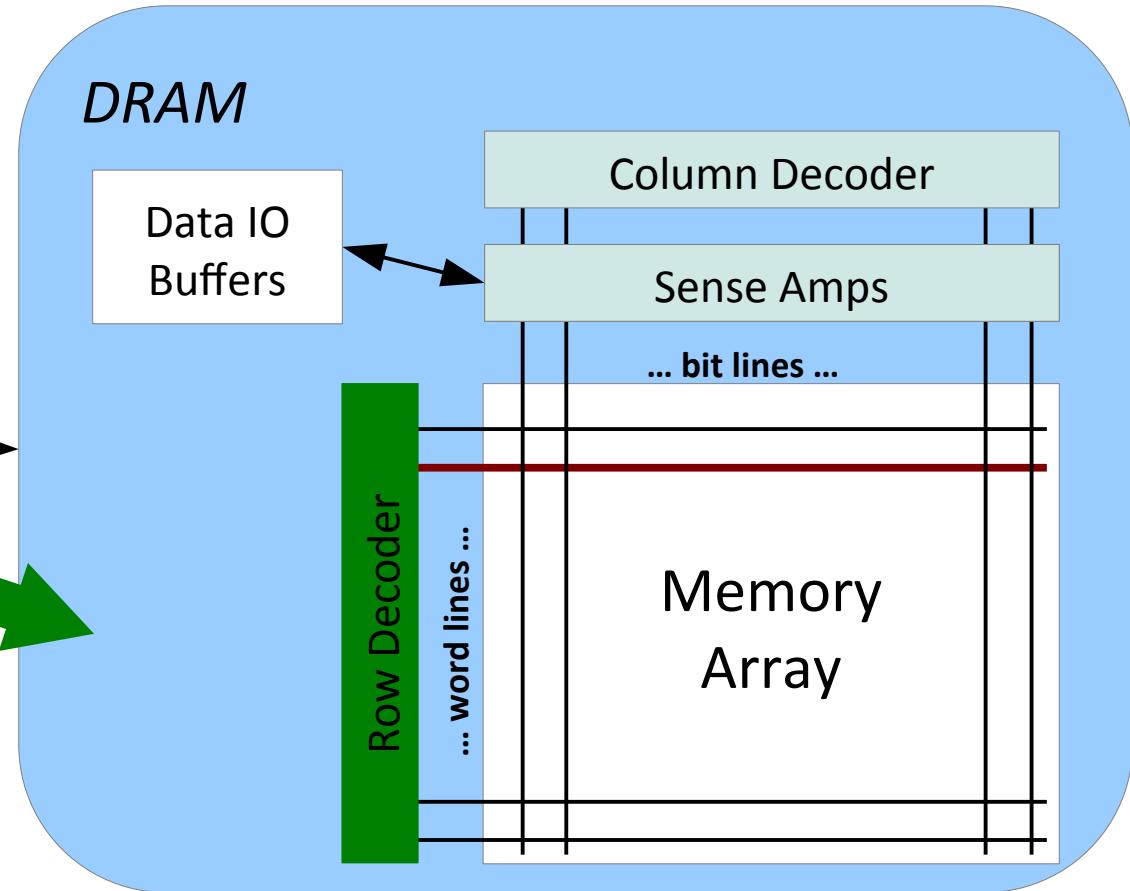
Протокол за достъп до DRAM

Колко бързо мога да достъпя
ред данни, считано от
заявката до презареждането



Известен като:

- ❖ Open a DRAM Page/Row или
- ❖ ACT (Activate a DRAM Page/Row)
- ❖ RAS (Row Access Strobe)



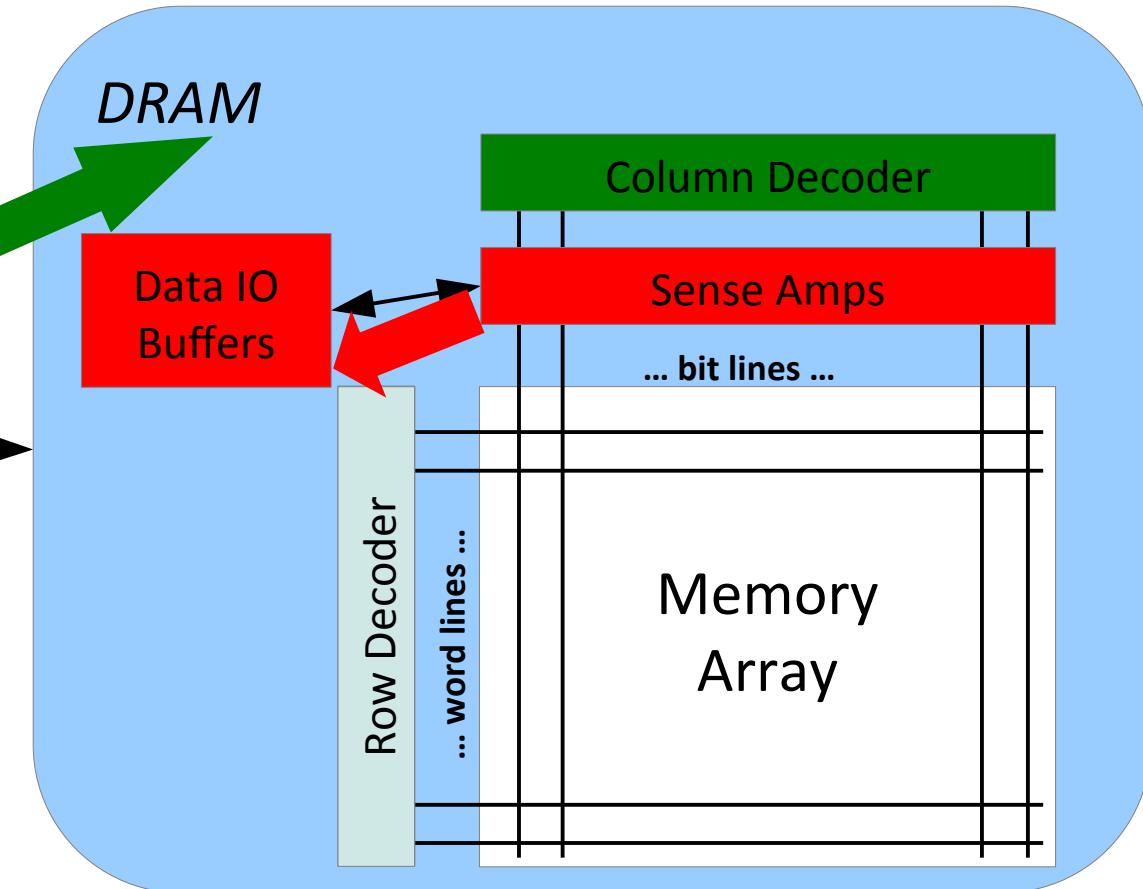
Протокол за достъп до DRAM

Колко бързо мога да достъпя
колона (най-малката
адресируема единица)



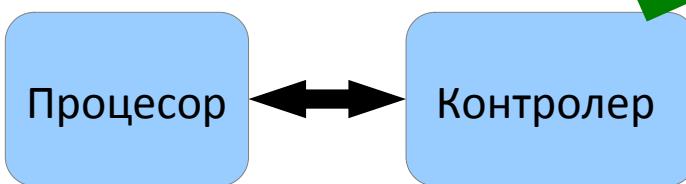
Известен като:

- ❖ READ Command
- ❖ CAS (Column Access Strobe)



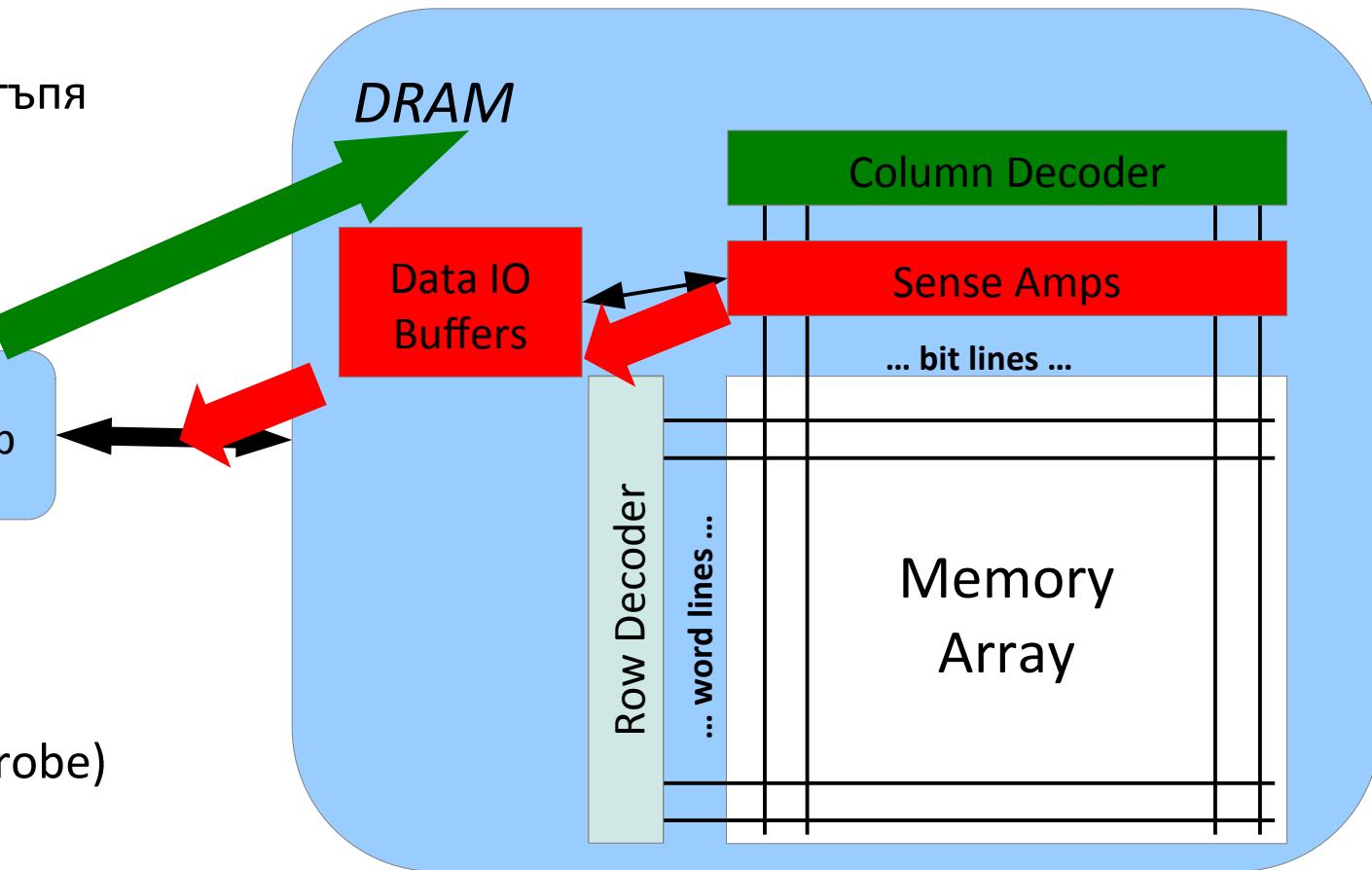
Протокол за достъп до DRAM

Колко бързо мога да достъпя
колона (най-малката
адресируема единица)

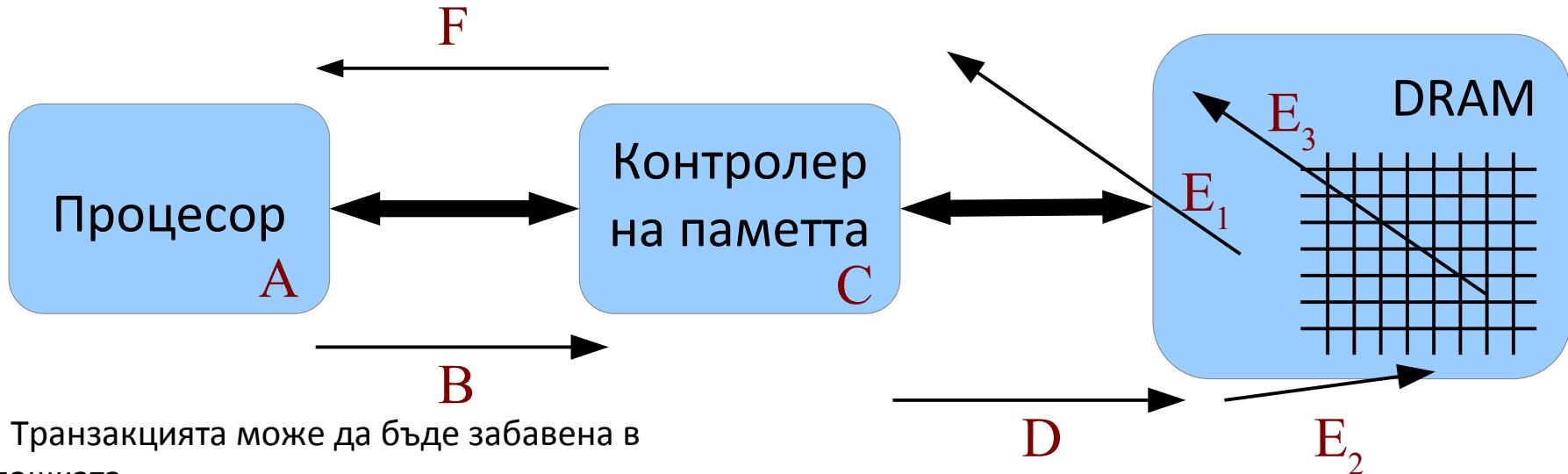


Известен като:

- ❖ CAS (Column Access Strobe)



Латентност при DRAM



- A: Транзакцията може да бъде забавена в опашката
- B: Транзакцията се изпраща до контролера
- C: Транзакцията се конвертира до последователност от команди (може да има опашка)
- D: Командите се изпращат до DRAM модула

- E₁: Има нужда от **CAS** или
- E₂: Има нужда от **RAS + CAS** или
- E₃: Има нужда от **PRE + RAS + CAS**

F: Транзакцията се връща на процесора

$$\text{Латентност} = A + B + C + D + E + F$$

Действие на DRAM

Три стъпки при четене/запис от дадена банка (по около 20 ns)

❖ Достъп до ред (RAS)

- ❖ Декодира адреса на реда, активира адресирания ред (често множество kB на ред)
- ❖ Бит линиите споделят заряда с клетката за съхранение
- ❖ Малка промяна в напрежението засечена от усилвателя маркира (latch) целия ред от битове
- ❖ Усилвателят кара бит линиите да презаредят клетката

❖ Достъп до колона (CAS)

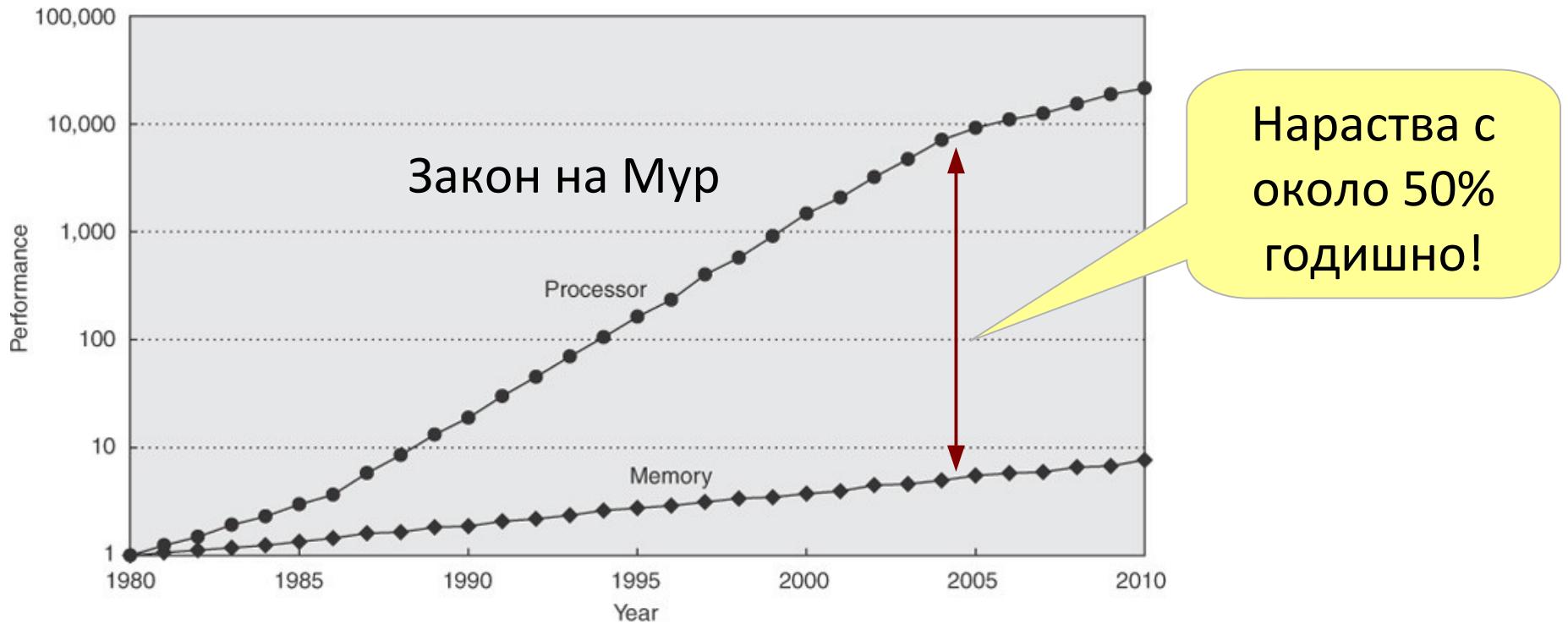
- ❖ Декодира се адреса на колоната, за да събере малък брой маркирани битове (4, 8, 16, 32, зависещ от DRAM пакета)
- ❖ При четене изпратените клетки се изпращат на изходните пинове
- ❖ При запис се презареждат маркираните битове с желаната стойност
- ❖ Може да има много достъпи до колони без друг достъп до ред (burst mode)

❖ Презареждане (Precharge)

- ❖ Зарежда бит линиите с междинна стойност преди следващ RAS



Производителност

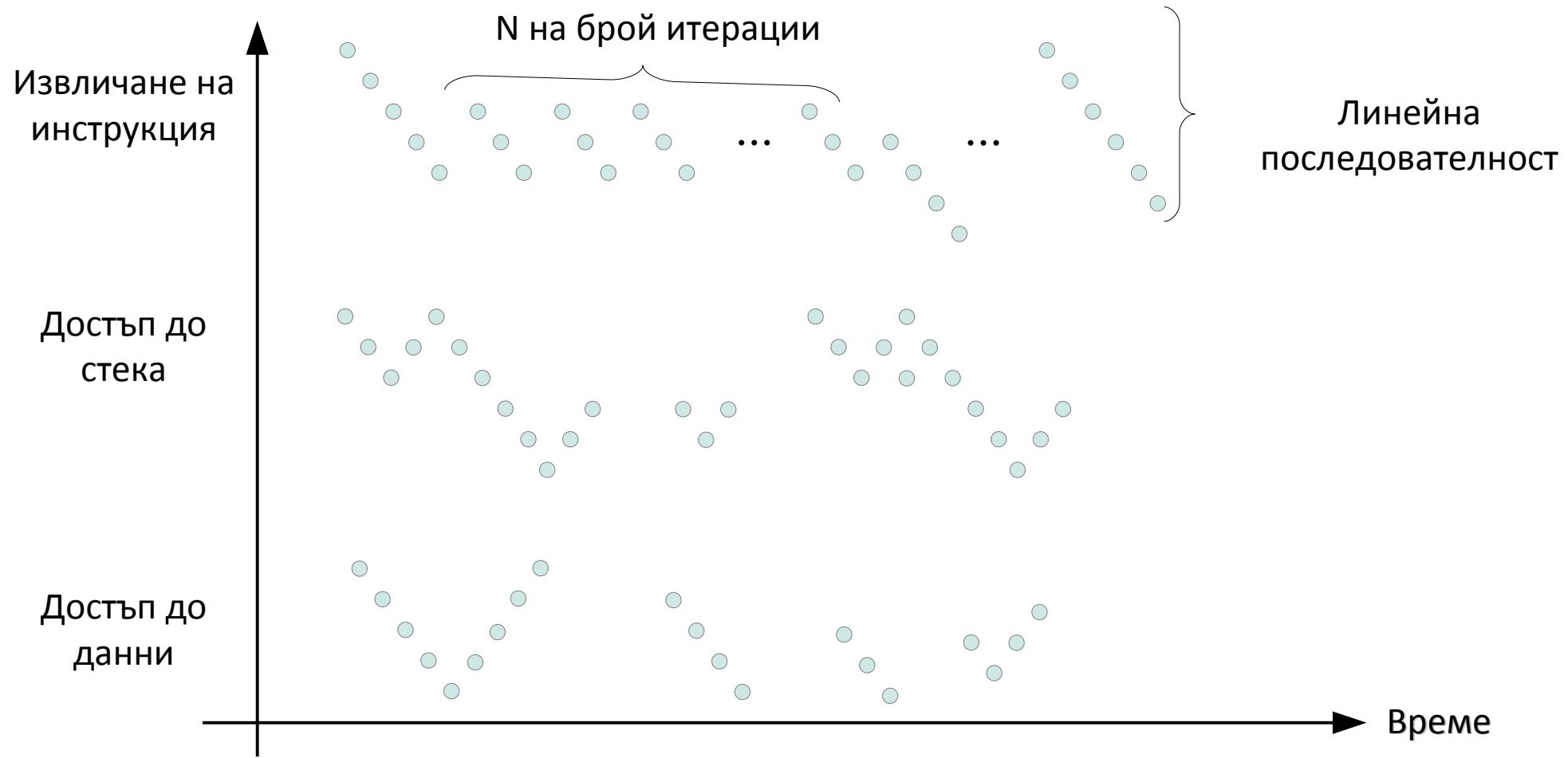


Памет на много нива

Идея: Прикриване на латентността чрез малки, бързи памети, наречени кешове

Кешовете са механизъм за прикриване на латентността, който се базира на емпиричното наблюдение, че начинът на достъп до паметта от процесора е добре предвидим.

Често срещани шаблони за достъп



Предвидими шаблони за достъп

Две предвидими свойства на достъпа до паметта:

- ❖ **Времева локалност (Temporal Locality)**

Ако се достъпи даден адрес, то е много вероятно да се достъпи пак в близко бъдеще

- ❖ **Пространствена локалност (Spatial Locality)**

Ако се достъпи даден адрес, то е много вероятно околните адреси да се достъпват в близко бъдеще

Йерархия от памет



- ❖ Размер: Регистър << SRAM << DRAM;
- ❖ Латентност: Регистър << SRAM << DRAM;
- ❖ Пропускателна способност: на чипа >> извън чипа;

При достъп до данни

- ❖ Попадение (hit) (данные са в быстрой памяти)
- ❖ Пропуск (miss) (данные не са в быстрой памяти)

Управление на йерархията от памет

Малка бърза памет, например регистри

- ❖ Адресът се съдържа в инструкцията
- ❖ Реализирана като съвкупност от регистри (*registry file*)

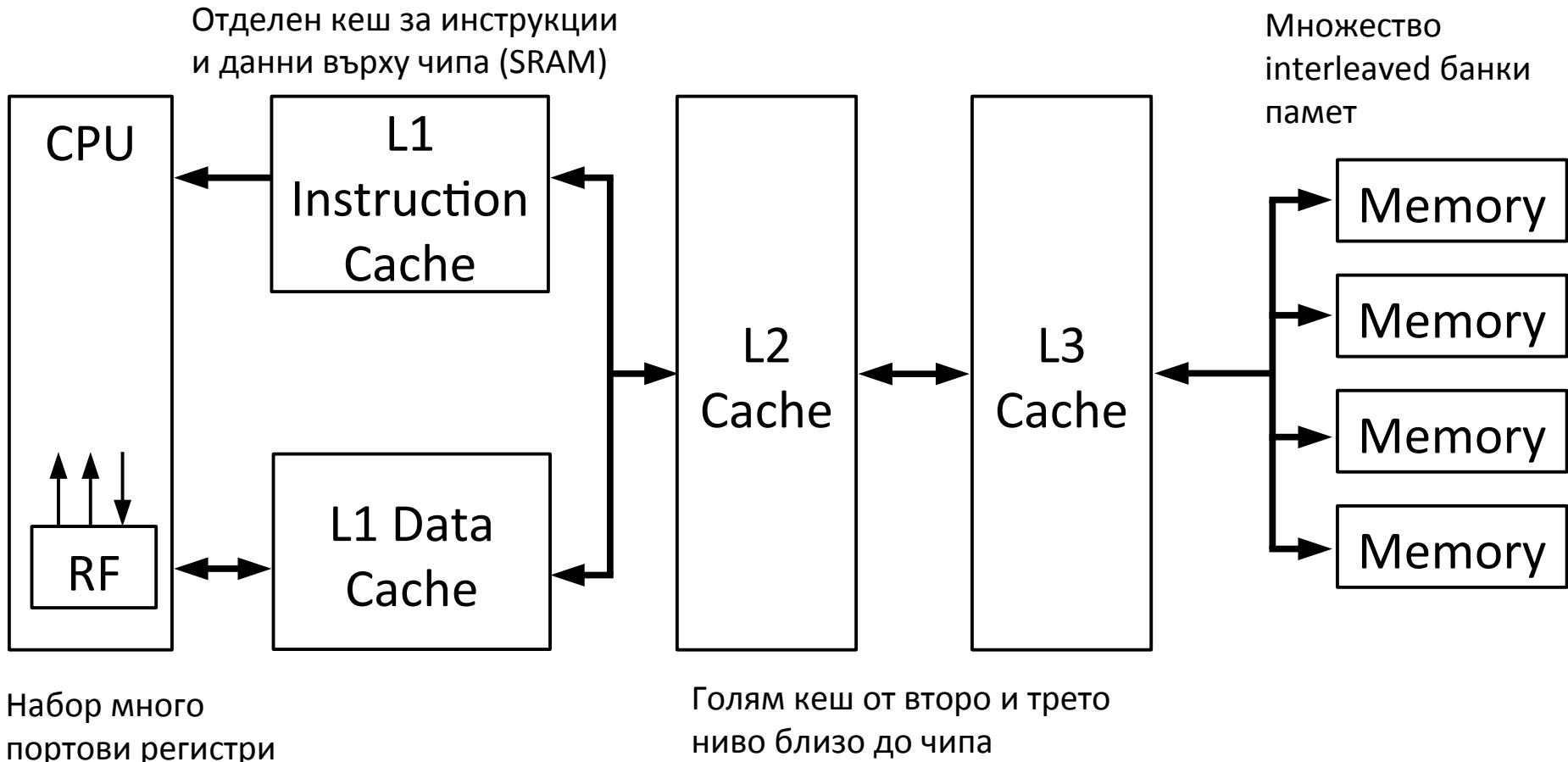
Хардуерът може да прави неща „зад гърба“ на софтуера, например управление на стека или преименуване на регистри

Голяма бавна памет

- ❖ Адресът се изчислява от стойностите в регистър
- ❖ Реализирана като йерархия от кешове

Хардуерът решава какво да съхрани в по-бързата памет. Софтуерът може да „подсказва“, например не кеширай или извлечи предварително

Типична йерархия от памети (2012)



Терминология при юерархията от памет

❖ Попадение (hit)

Данните са в някой блок на бързата памет

❖ Процент попадения (hit rate)

Съотношението на достъпа до бързата памет и общия брой достъпи

❖ Време на попадение (hit time)

Времето, което отнема да се извлече заявката в бързата памет. Състои се от време за достъп до бързата памет и време за определяне дали е попадение или пропуск

❖ Пропуск (miss)

Данните трябва да се извлекат от по-бавната памет

❖ Процент пропуски (miss rate)

Процент пропуски = 1 – процентът попадения

❖ Наказание при пропуск (miss penalty)

Времето, за което се замества блок в бързата памет с намерения блок и се доставя до проц

❖ Времето на попадение << Наказанието при пропуск



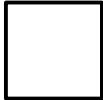
Проблеми при юерархията от памет

- ❖ Идентификация на блок
 - ❖ Как се намира блокът, ако е в по-близката (по-бърза) памет?
Tag/Block
- ❖ Слагане на блок
 - ❖ Къде може да се сложи блок в по-близката (по-бързата) памет?
Пълна асоциативност, Множествена асоциативност, Директно съпоставяне
- ❖ Заместване на блок
 - ❖ Кой блок трябва да се замени при пропуск?
Произволен, LRU
- ❖ Стратегия за запис
- ❖ Какво става при запис
Write back, Write through (с Write buffer)



Кеш, използващ директно съпоставяне

Бит за валидност



Кеш таг



Кеширани данни

Байт 3

Байт 2

Байт 1

Байт 0

❖ Ефект Пинг Понг

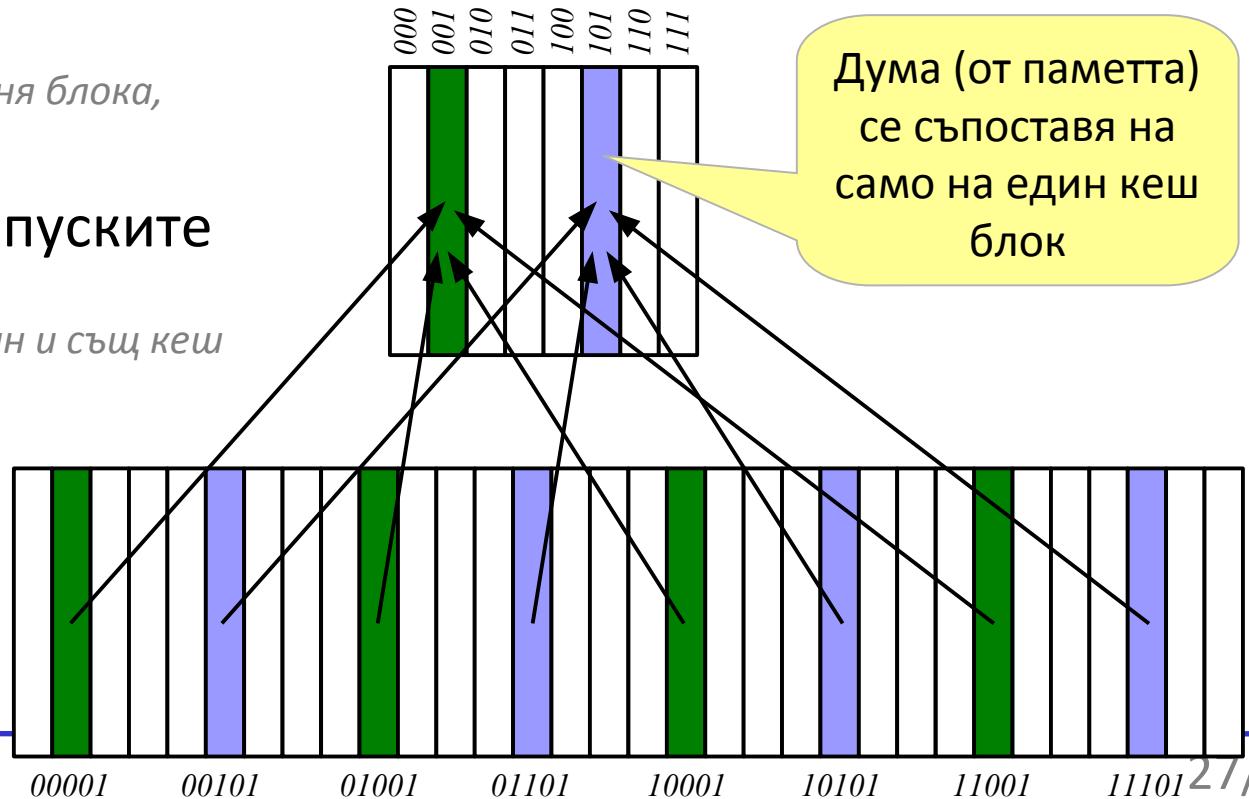
Най-лошият случай е да се заменя блока, последван от пропуск

❖ За да се намалят пропуските

Увеличаваме размера на кеша

Множество стойности за един и същ кеш индекс

Адресът в кеша =
(Адресът на блока) mod
(Броят кеш блокове)



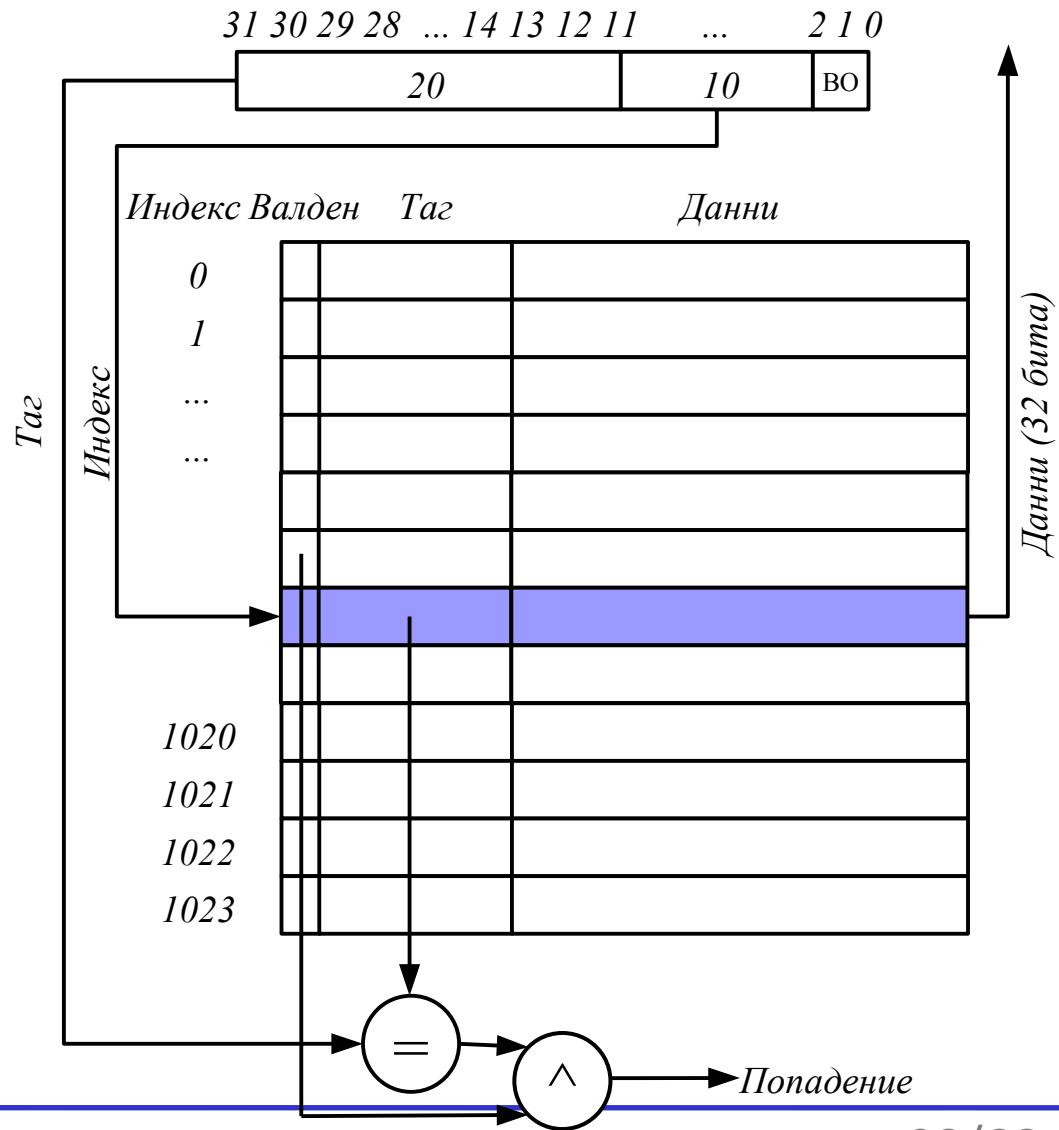
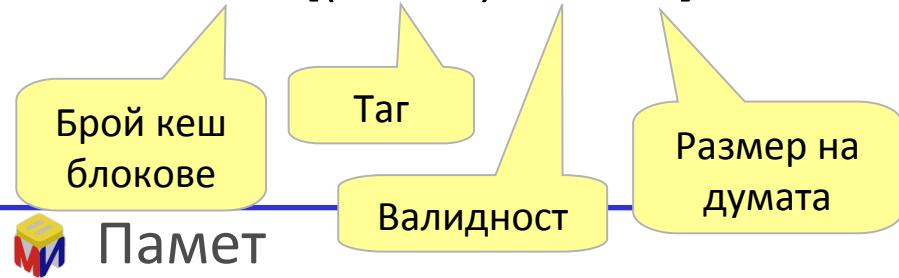
Достъп до кеша

Размерът на кеша зависи от

- ❖ Броя кеш блокове
- ❖ Броя битове в адреса
- ❖ Размера на думата

Например

- ❖ При n-битов адрес, 4-байтова дума и 1024 кеш блока
- ❖ Размерът на кеша = $1024[(n-10-2) + 1 + 32]$ бита

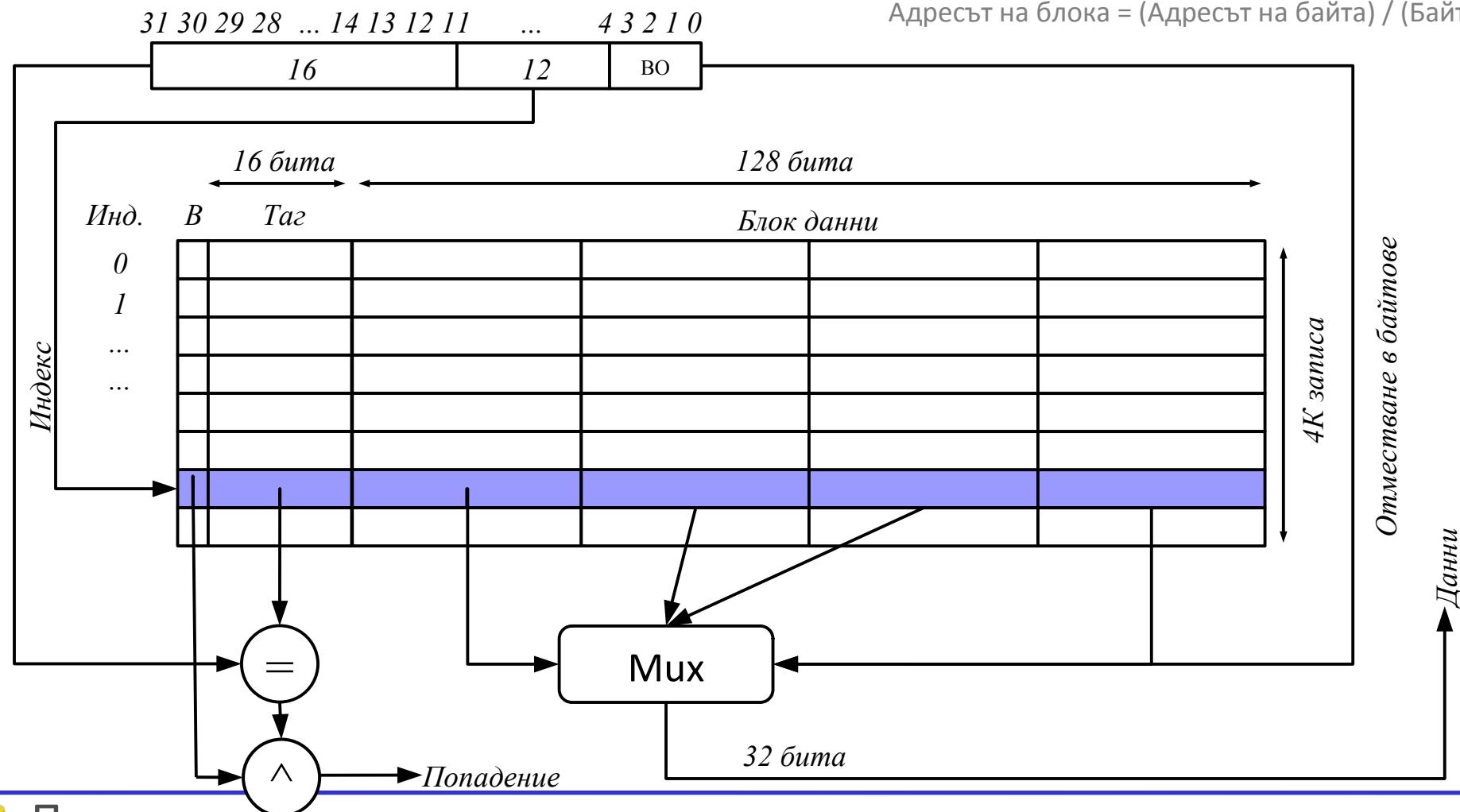


Достъп до кеша

Възползва се от пространствената локалност

Адресът в кеша = (Адресът на блока) mod (Броят кеш блокове)

Адресът на блока = (Адресът на байта) / (Байтове на блок)



Определяне на размера на блоковете

По-големи блокове данни се възползват по-добре от пространствената локалност, НО:

- ❖ По-големи блокове означава по-голямо наказание при пропуск
Повече време за запълване на блок
- ❖ Ако размерът на блока е прекалено голям в сравнение с размера на кеша, то процентът пропуски се повишава
Прекалено малко кеш блокове (блокове от данни в кеша)

Средното време за достъп =

Времето за попадение * (1 – Процентът пропуски) +
Наказанието за пропуск * Процентът пропуски

Местоположение на блоковете

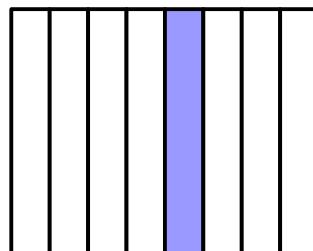
Хардуерна сложност

Оползтвояване на кеша

Кеш с директно съпоставяне

Номер блок: 0 1 2 3 4 5 6 7

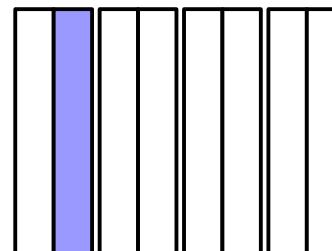
Данни



Кеш с множествена асоциативност

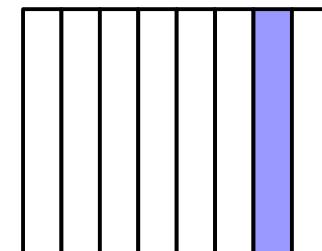
Номер масив: 0 1 2 3

Данни

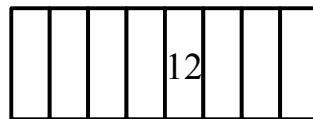


Кеш с пълна асоциативност

Данни



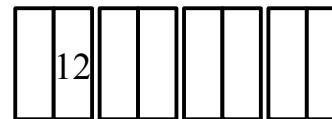
Търсене



Блок 12 може да
бъде поставен:

$$12 \bmod 8 = 4$$

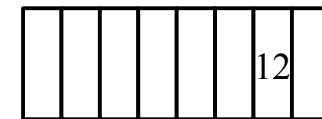
Търсене



Търсене

$$12 \bmod 4 = 0$$

Търсене



- ❖ Номерът на масив = (Номерът на блок) mod (Броя на масиви в кеша)
- ❖ Повишената гъвкавост на поставянето на блокове намалява вероятността от пропуски

Управление на пропуските

- ❖ Пропуски при четене винаги извлича блокове от паметта – Политика на заместване;
- ❖ При запис трябва внимателна поддръжка на консистентността на кеша и оперативната памет (ОП) – Политики на запис;

Видове пропуски

❖ Студен пропуск

При първи достъп до данните

- ❖ Предварително извлечане може да намали забавянето

❖ Пропуск породен от капацитета

*Предходният запис е бил премахнат, защото прекалено много други данни са били достъпвани.
Множеството работни данни е прекалено голямо*

- ❖ Реорганизиране на алгоритъма да преизползва данните преди да се премахнат от кеша
- ❖ В противен случай предварително извлечане

Видове пропуски

❖ Пропуск при конфликти

Много данни се съпостават на едно и също място в кеша, което довежда до премахване дори и когато кеша не е пълен

❖ Реорганизиране на данните и/или изместване на масивите

❖ Пропуск True Sharing

Нишка в друг процесор е използвала данните и те са преместени в друг кеш

❖ Минимизиране на споделянето/заключването

❖ Пропуск False Sharing

Другия процесор е използвал други данни в същия запис в кеша и записа е бил преместен

❖ Изместване (падинг) на данните и осигуряване на заключени структури да не попадат в кеша

Политики на заместване на ел. от кеша

При асоциативен кеш, кой блок памет трябва да се замени когато множеството се напълни?

- ❖ Произволен;
- ❖ Най-рядко използваният (Least Recently Used)
 - ❖ Състоянието трябва да бъде обновявано при всеки достъп
 - ❖ Реализацията има смисъл при кешове с малки множества на асоциативност (2, 4 мн.)
 - ❖ Псевдо LRU (като двоично дърво или чрез битове) (4, 8 мн.)
- ❖ First In First Out т.нр. Round-Robin (високо асоц. Кешове);
- ❖ Not Least Recently Used (NLRU)
 - ❖ FIFO с изключение за най-често използвания блок



Политики на заместване на ел. от кеша

❖ Тривиално при директното съпоставяне

Асоциативност Размер	2 асоциативен		4 асоциативен		8 асоциативен	
	LRU	Произв.	LRU	Произв.	LRU	Произв.
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Емпиричните резултати показват, че с увеличаване на размера на кеша се намалява значението на политиката на заместване

Политики за запис

- ❖ При попадение
 - ❖ Write through: запис в кеша и ОП
По-голям трафик, но по-лесна синхронизация и кохерентност на кеша с паметта
 - ❖ Write back: запис само в кеша
Записва се в паметта само когато блока се извади от кеша. Поддържа се „мръсен“ (dirty) бит, чрез който се намалява трафика
- ❖ При пропуск
 - ❖ No write allocate: запис само в ОП
 - ❖ Write allocate (aka извличане при запис)
- ❖ Хибридни
 - ❖ Write through and no write allocate
 - ❖ Write back with write allocate

Прост кеш. Начални условия.

- ❖ 32Кб, директно съпоставяне, 64 байтови линии (512 записи);
- ❖ Достъп до кеша – един цикъл;
- ❖ Достъп до паметта – 100 цикъла;
- ❖ Думата е 1 байт;

Начини на достъп. Примери

```
#define S ((1<<20)*sizeof(int))  
int A[S];  
  
for(i=0; i<S; i++)  
    Read A[i];
```



Достъп до данните:

- ❖ Четене на нов запис
- ❖ Четене на останалата част от записа
- ❖ Преместване на следващия запис

`sizeof(int) = 4`

16 елемента на запис(линия)

15 от всеки 16 са попадения

Общо време за достъп: $15*S/16 + 100*S/16$

Какъв вид локалност? Пространствена

Какви видове пропуски? Студени

Начини на достъп. Примери

```
#define S ((1<<20)*sizeof(int))  
int A[S];  
  
for(i=0; i<S; i++)  
    Read A[0];
```



Достъп до данните:

❖ Четене A[0] всеки път

$\text{sizeof}(\text{int}) = 4$

S броя четене

Всички достъпи (без първия) са попадения

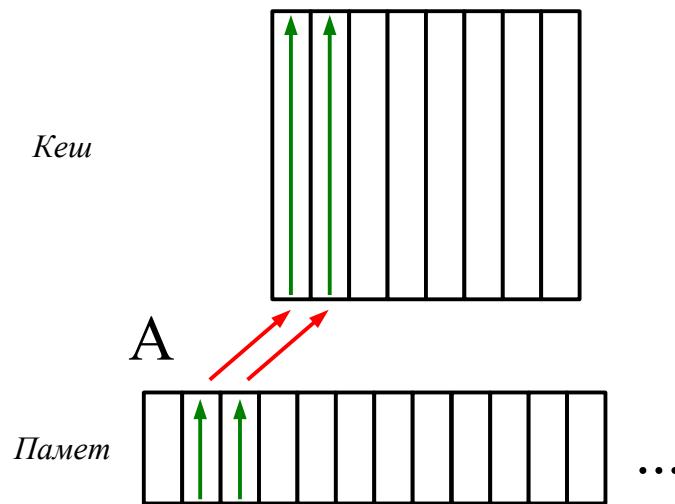
Общо време за достъп: $100 + S - 1$

Какъв вид локалност? Времева

Какви видове пропуски? Студени

Начини на достъп. Примери

```
#define S ((1<<20)*sizeof(int))  
int A[S];  
  
for(i=0; i<S; i++)  
    Read A[i % (1<<N)];
```



Достъп до данните:

- ❖ Четене на началния сегмент на A много пъти

S броя четене

$4 \leq N \leq 13$

Един пропуск за всеки достъпен запис.
Останалите са попадения.

Достъпени записи: 2^{N-4}

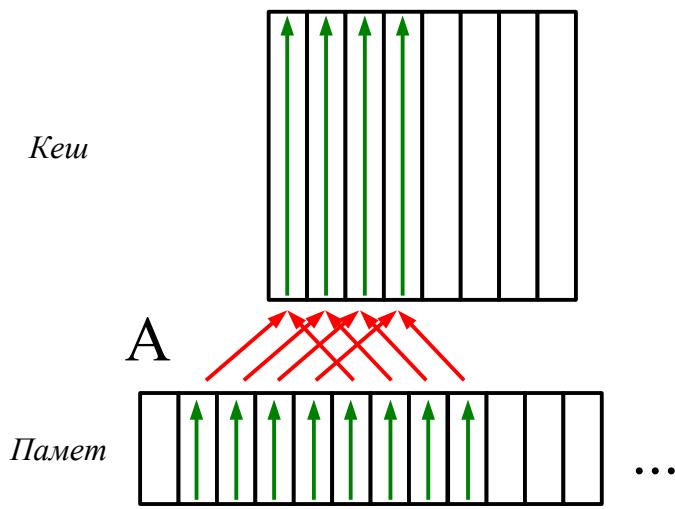
Общо време за достъп: $2^{N-4} * 100 + S - 2^{N-4}$

Какъв вид локалност? Времева,
Пространствена

Какви видове пропуски? Студени

Начини на достъп. Примери

```
#define S ((1<<20)*sizeof(int))  
int A[S];  
  
for(i=0; i<S; i++)  
    Read A[i % (1<<14)];
```



Достъп до данните:

- ❖ Четене на началния сегмент на A много пъти

S броя четене

N>=14

Един пропуск за всеки достъпен запис.
Останалите са попадения.

Общо време за достъп: $15*S/16 + 100*S/16$

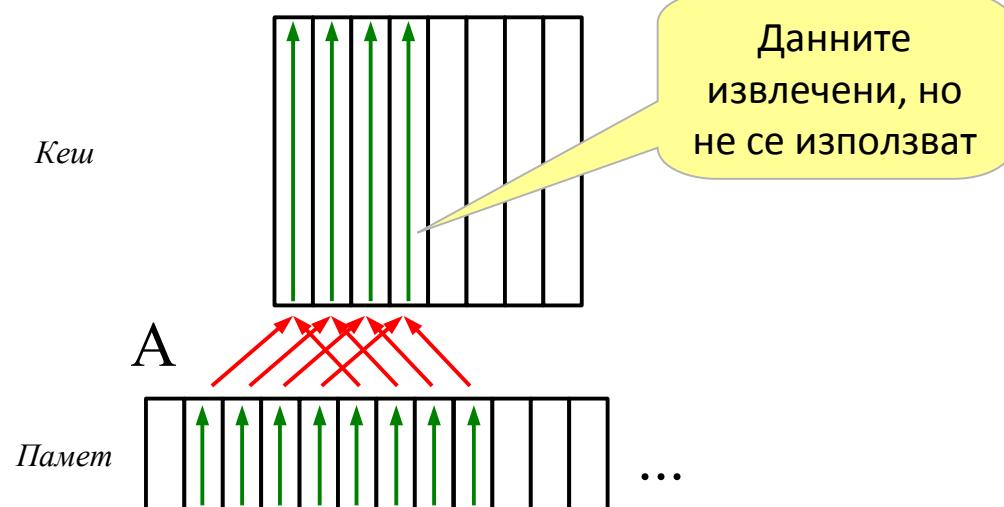
Какъв вид локалност?

Пространствена

Какви видове пропуски? Студени, в
капацитета

Начини на достъп. Примери

```
#define S ((1<<20)*sizeof(int))  
int A[S];  
  
for(i=0; i<S; i++)  
    Read A[(i*16) % (1<<14)];
```



Достъп до данните:

- ❖ Четене на всеки 16-ти елемент от началния сегмент на А много пъти

S броя четене

N>=14

Първи достъп до запис – пропуск.

Общо време за достъп: 100*S

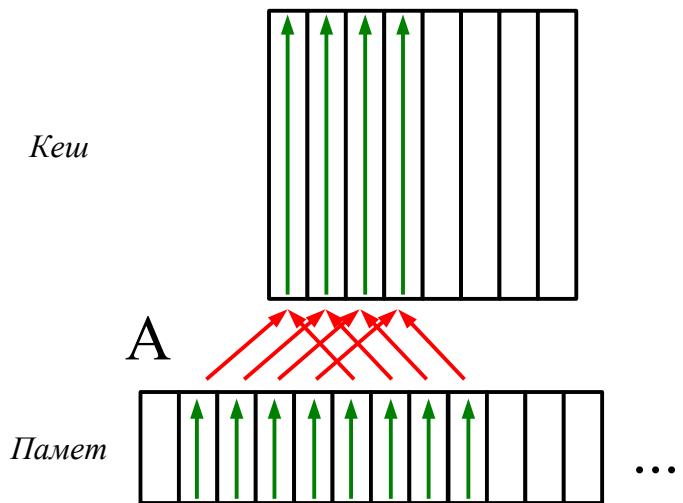
Какъв вид локалност? Никаква

Какви видове пропуски?

Студени, в капацитета

Начини на достъп. Примери

```
#define S ((1<<20)*sizeof(int))  
int A[S];  
  
for(i=0; i<S; i++)  
    Read A[random() %S];
```



Достъп до данните:

- ❖ Четене на произволни елементи от А
S броя четене

Вероятност за попадение в кеша=8Кб/1Гб
= 1/256

Общо време за достъп:

$$S * 255/256 * 100 + S * 1/256$$

Какъв вид локалност?

Почти никаква

Какви видове пропуски?

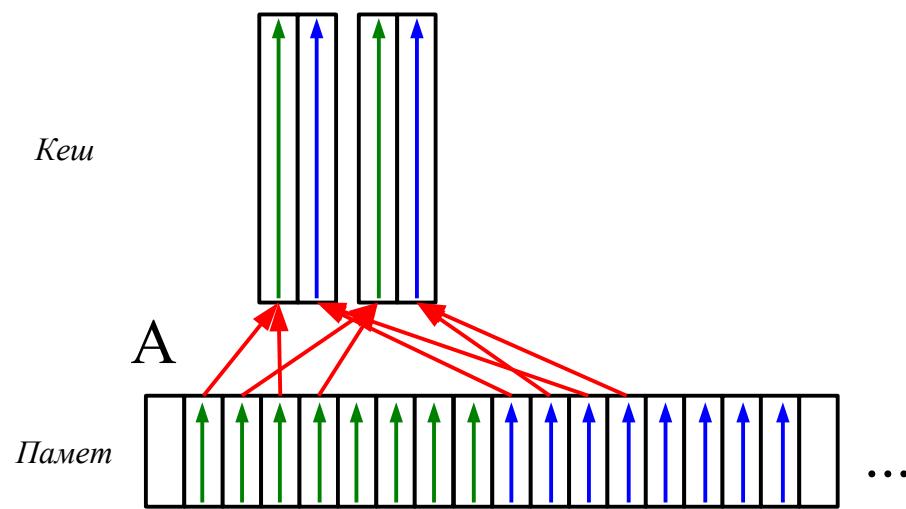
Студени, в капацитета, конфликти

Множествено асоц. кеш. Начални условия.

- ❖ 32Кб, двуасоциативен, 64 байтови линии 256 множества;
- ❖ Много линии на множество, наричани пътища (way);
- ❖ Всяка дума от паметта се съпоставя на специфичен масив;
- ❖ Думата е 1 байт;

Начини на достъп. Примери

```
#define S ((1<<19)*sizeof(int))  
int A[S];  
int B[S];  
  
for(i=0; i<S; i++)  
    Read A[i], B[i];
```



Достъп до данните:

- ❖ Четене на елементи от А и В последователно

S броя четене

А и В се презастьпват в един и същи път, но има достатъчно линии

Общо време за достъп:

$$2*(15/16*S + 1/6*S*100)$$

Какъв вид локалност?

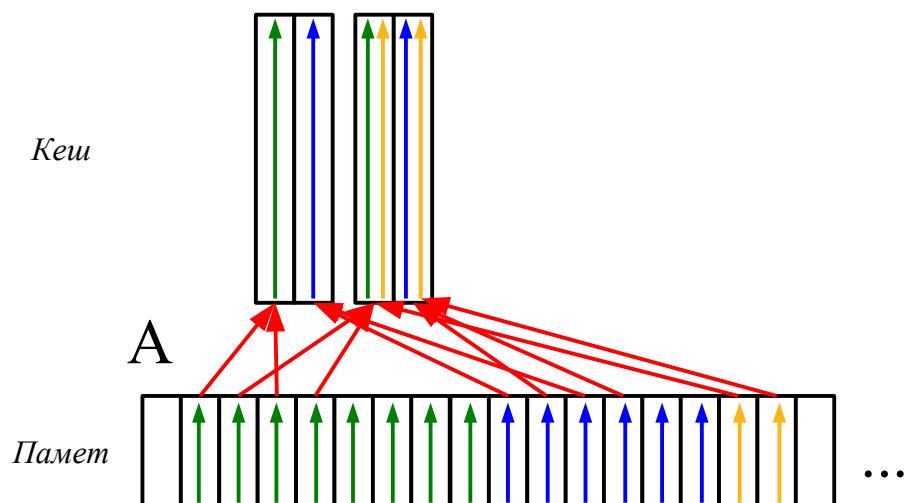
Пространствена локалност

Какви видове пропуски?

Студени

Начини на достъп. Примери

```
#define S ((1<<19)*sizeof(int))  
int A[S];  
int B[S];  
int C[S];  
  
for(i=0; i<S; i++)  
    Read A[i], B[i], C[i];
```



Достъп до данните:

- ❖ Четене на елементи от А и В последователно

S броя четене

А и В се презастьпват в един и същи път, но няма достатъчно линии

Общо време за достъп (с LRU):

$3*S*100$

Какъв вид локалност?

Пространствена локалност

Какви видове пропуски?

Студени, конфликти

Пример

Блоково умножение на Матрицы

L1D кеш: 32 КБ
L2 кеш: 6 МБ
sizeof(double) = 8 байта

1. $3 \cdot 16 \cdot 16 \cdot 8 = 6 \text{ КБ}$

L1 L2

2. $3 \cdot 32 \cdot 32 \cdot 8 = 24 \text{ КБ}$

L1 L2

3. $3 \cdot 64 \cdot 64 \cdot 8 = 96 \text{ КБ}$

L1 L2

4. $3 \cdot 128 \cdot 128 \cdot 8 = 384 \text{ КБ}$

L1 L2

5. $2 \cdot 256 \cdot 256 \cdot 8 = 1.5 \text{ МБ}$

L1 L2

6. $3 \cdot 512 \cdot 512 \cdot 8 = 6 \text{ МБ}$

L1 L2

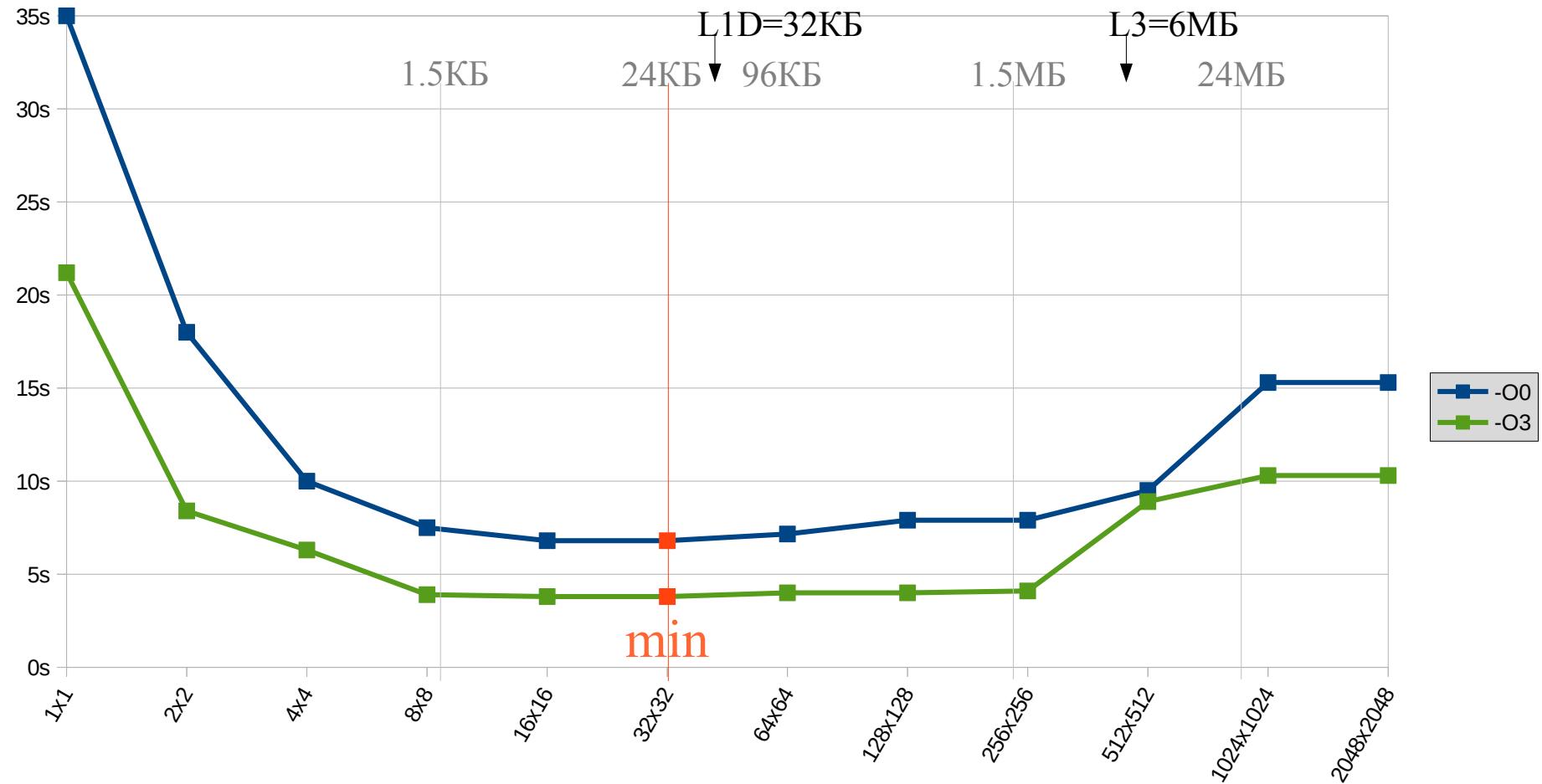
7. $3 \cdot 1024 \cdot 1024 \cdot 8 = 24 \text{ МБ}$

L1 L2

8. $3 \cdot 2048 \cdot 2048 \cdot 8 = 32 \text{ МБ}$

L1 L2

Матрицы 1024×1024 на i7 (32КБ, 256КБ, 6МБ)



Архитектури на Паметта на Паралелните Компютри



Паралелни архитектури

❖ Споделена памет (Shared memory)

Много процесори могат да правят достъп до обща памет.

❖ Еднороден достъп до паметта (Uniform memory access – UMA)

Идентични процесори имат еднакъв достъп и еднако време на достъп до паметта.

❖ Не-еднороден достъп до паметта (Non-uniform memory access – NUMA)

Не всички процесори имат еднакъв достъп и еднако време на достъп до паметта.

❖ Cache only memory architecture (COMA)

Цялата памет на възлите се използва само като кеш.

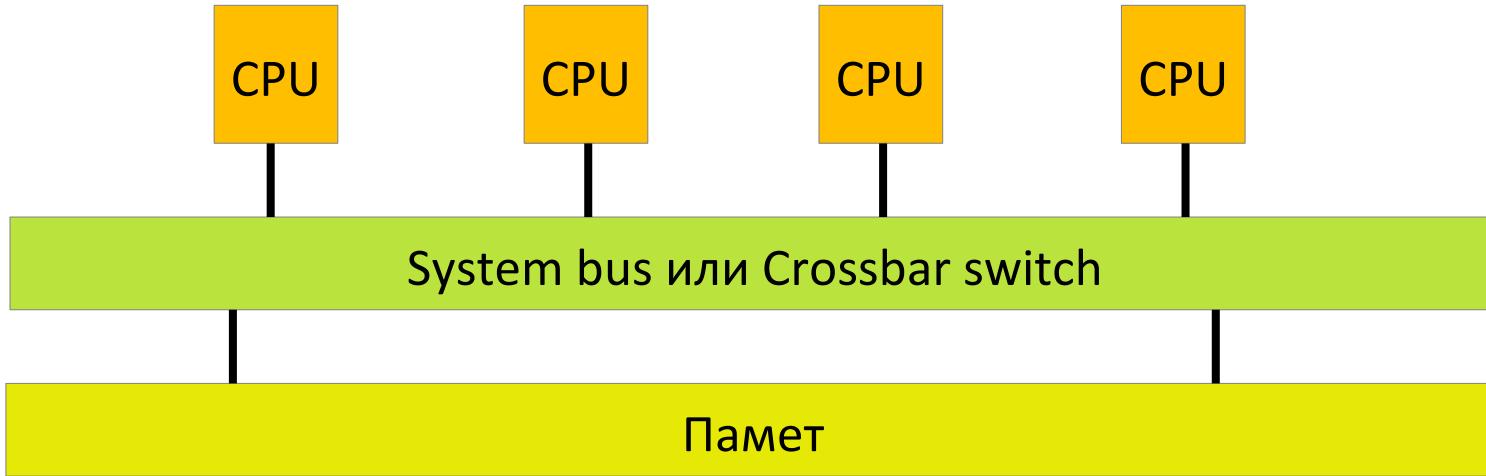
❖ Разпределена памет (Distributed memory)

Изиска комуникация по мрежата за достъп до между процесорната памет.

❖ Хибридна Разпределено-Споделена памет (Hybrid Distributed-Shared Memory)

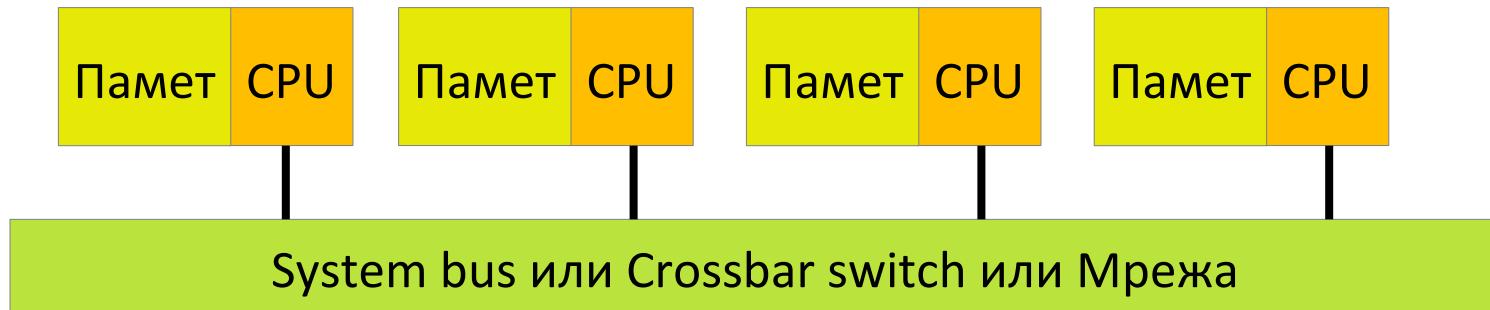


Споделена памет (Shared memory)



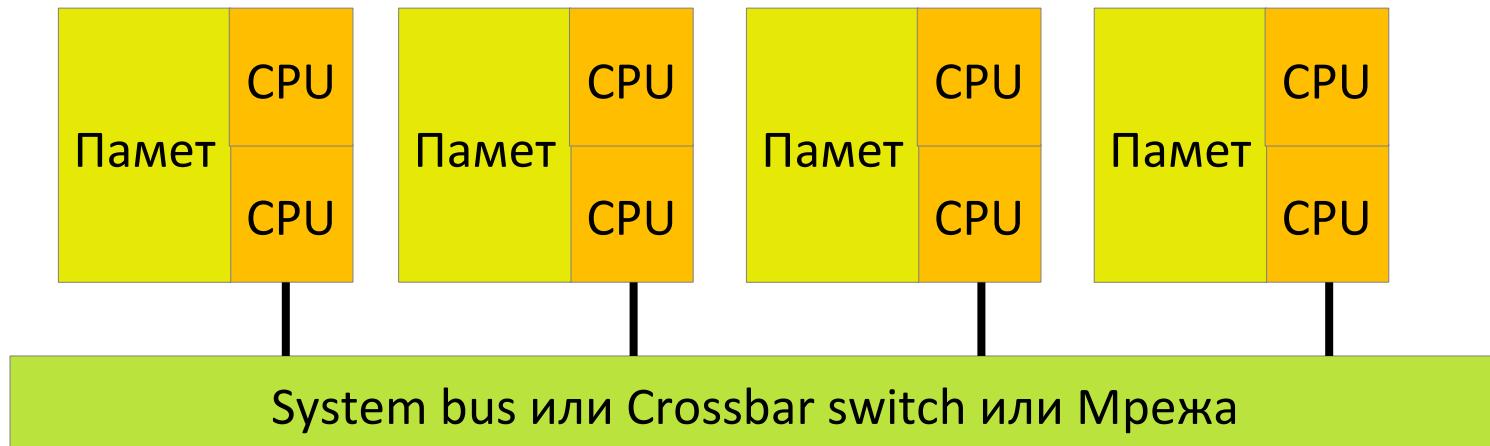
Всички процесори могат да правят достъп до обща памет.

Разпределена памет (*Distributed memory*)



Изисква комуникация (по мрежата) за достъп до разпределената памет.

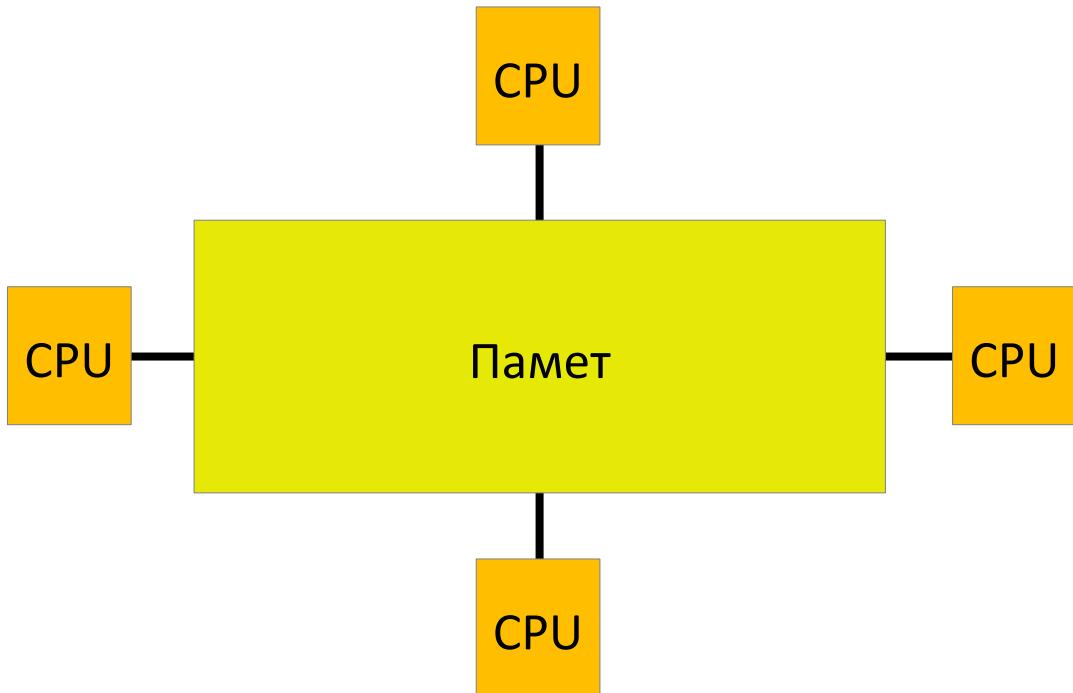
Хибридна Разпределено-Споделена памет (Hybrid Distributed-Shared Memory)



Притежава повечето от предимствата и недостатъците на предишните две.
Това е най-използваната в момента архитектура в големите компютри.

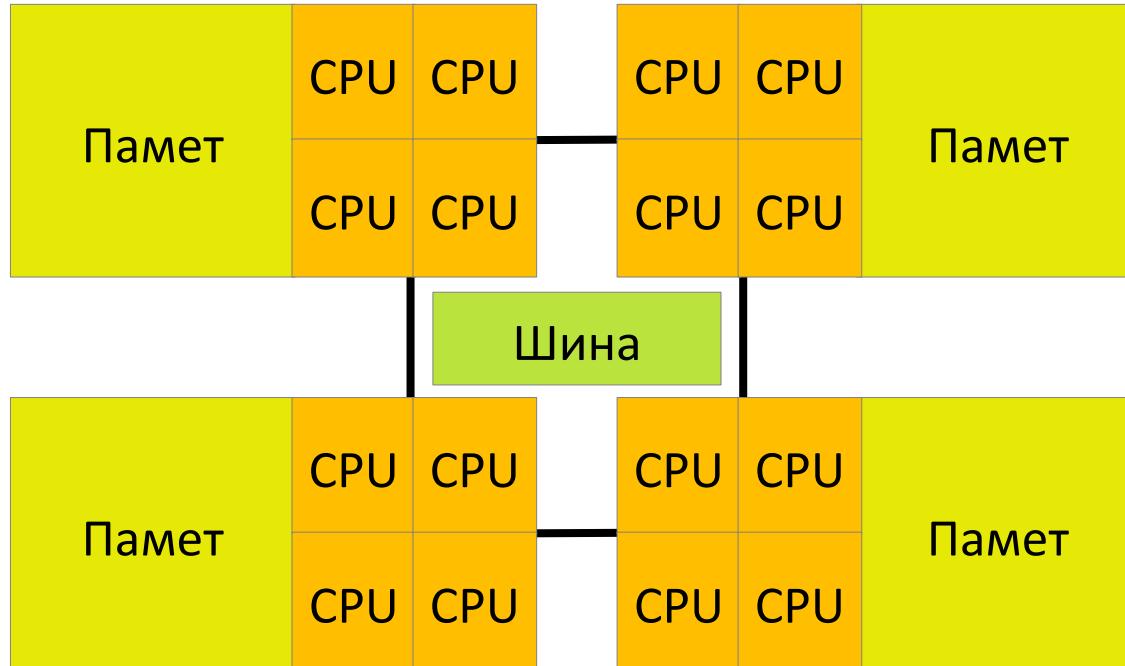
Еднороден достъп до паметта (Uniform memory access – UMA)

(SM)



- ❖ Много (идентични) процесори;
- ❖ Равноправен достъп;
- ❖ Еднакво време за достъп;

Не-еднороден достъп до паметта (Non-uniform memory access – NUMA)

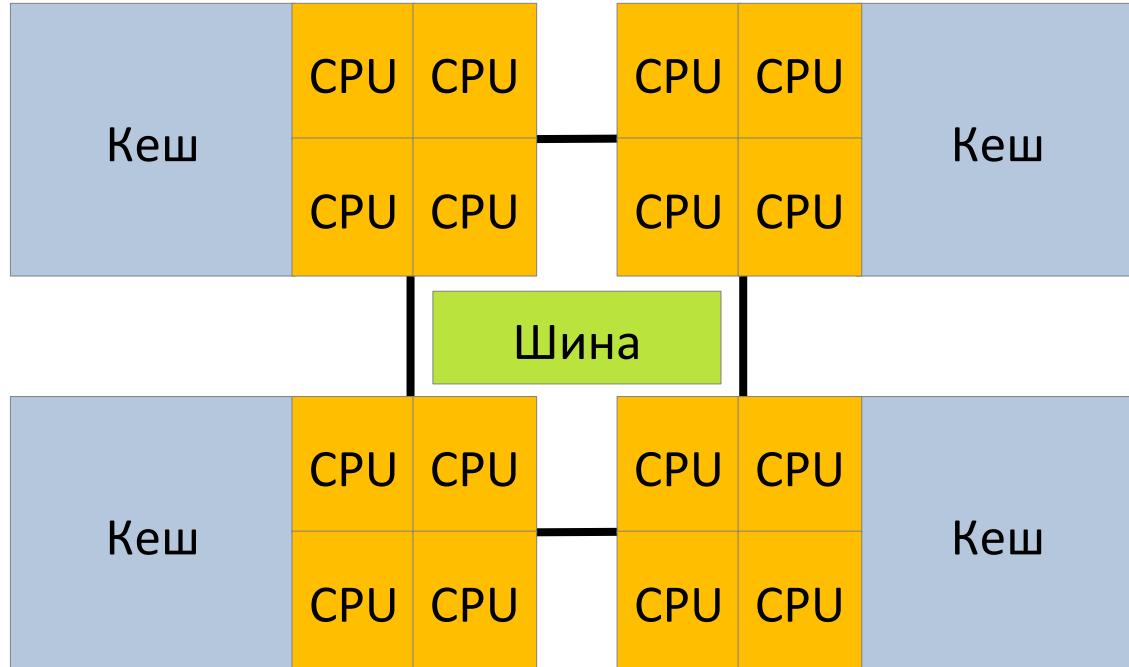


- ❖ Много процесори;
- ❖ Не всички процесори имат равноправен достъп до цялата памет;
- ❖ Достъпа до паметта през връзката е бавен;
- ❖ Програмиста гарантира за коректния достъп до глобалната памет;

Само кеш

(SM)

(Cache only memory architecture – COMA)



- ❖ Много процесори;
- ❖ Цялата памет се използва за кеш на данните;
- ❖ Може да се използва съвместно/хибридно с NUMA;

Shared-Nothing Архитектура

Shared-Nothing Архитектура

- ❖ Архитектурата Shared-Nothing (SN) е разпределена архитектура, в която всяка заявка за актуализация се изпълнява от един възел (процесор/памет/диск). Целта е да се премахне “съперничеството” между възлите; (*Web, DB, ...*)
- ❖ Възлите не споделят (едновременен достъп до) памет или диск;
- ❖ SN елиминира единични точки на повреда, което позволява на цялостната система да продължи да работи въпреки грешките в отделните възли;
- ❖ SN може да се мащабира чрез лесно добавяне на възли;
- ❖ SN обикновено разпределя данните между много възли, чрез репликация, което позволява да се изпълняват повече заявки;

Масивно Паралелни Компютри

GRID Системи,

Компютърни Кълстъри,

Силно паралелни процесорни масиви (MPPA)

Масивно Паралелни Компютри

Масивно паралелни компютри е използването на голям брой процесори (или отделни компютри) за извършване на набор от координирани изчисления паралелно (едновременно).

- ❖ GRID системи/изчисления;
- ❖ Компютърни клъстери;
- ❖ Силно паралелни процесорни масиви (МРРА);

GRID Системи

Grid системите са широко разпределени компютърни ресурси използвани заедно за постигането на обща цел.

- ❖ Използват middle-ware (например BOINC и др.);
- ❖ Те са форма на разпределен компютър (супер виртуален компютър);
- ❖ Изискват мрежа (Интернет) за връзка между системите;
- ❖ Слабо свързани системи;
- ❖ Обикновено силно хетерогенни;

Например, Worldwide LHC Computing Grid (WLCG) на ЦЕРН – над 170 компютърни центъра в 42 държави; Над 200000 ядра;



Компютърни Клъстери

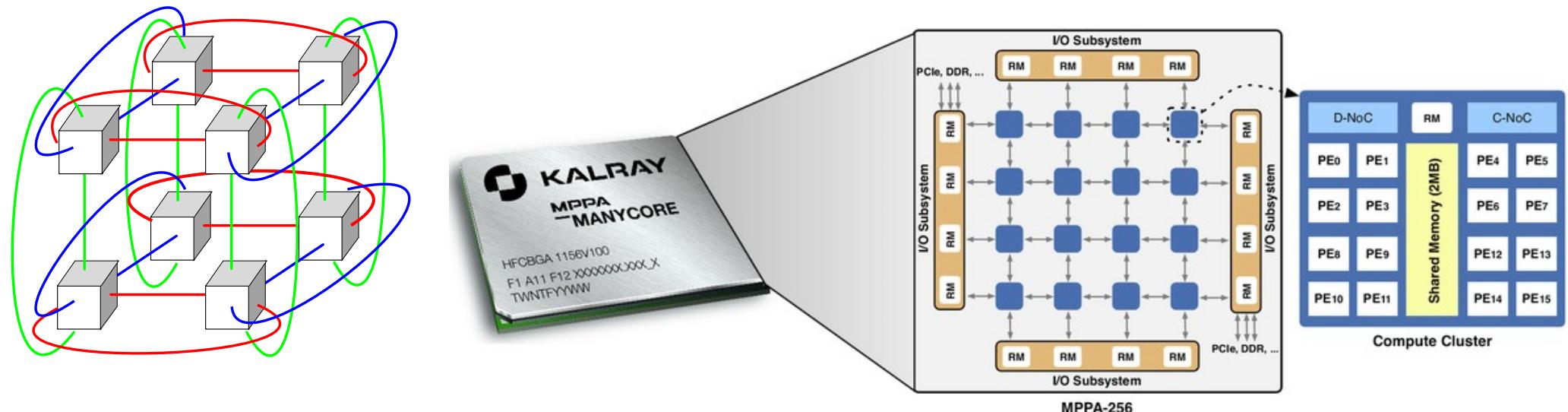
Компютърен клъстер се нарича множество от компютри, което е (тясно) свързано за изпълнение на една задача и може да се разглежда като една цялостна система.

- ❖ Изискват (локална) мрежа, InfiniBand връзка и др.;
- ❖ Тясно свързани системи;
- ❖ Обикновено сравнително хомогенни;
- ❖ Специализиран софтуер оркестрира работата на клъстера;



Силно паралелни процесорни масиви (MPPA)

MPPA са вид интегрални схеми, които представляват масив от стотици или хиляди процесори и RAM памет. Тези процесори комуникират по между си чрез преконфигурируема взаимосвързаност на каналите.



Въпроси?

apenev@uni-plovdiv.bg



Паралелно Програмиране

Видове паралелизъм.

Векторизация.

Задачи – фибри, нишки, процеси...

Видове многозадачност.

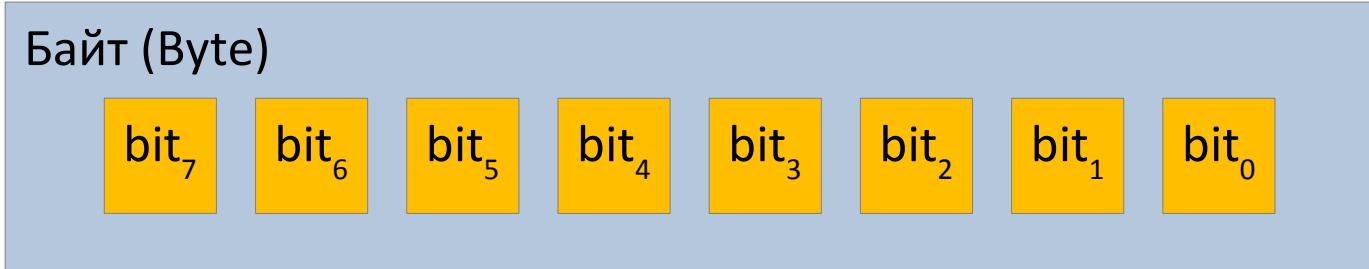
доц. д-р Александър Пенев

Видове Паралелизъм

Видове

- ❖ Bit-Level Паралелизъм (BLP);
- ❖ Instruction-Level Паралелизъм (ILP);
- ❖ Високо ниво:
 - ❖ Task-Level Паралелизъм (TLP, Функционален паралелизъм);
 - ❖ Data-Level Паралелизъм (DLP, Даннов паралелизъм);

Bit-Level Паралелизъм (BLP)

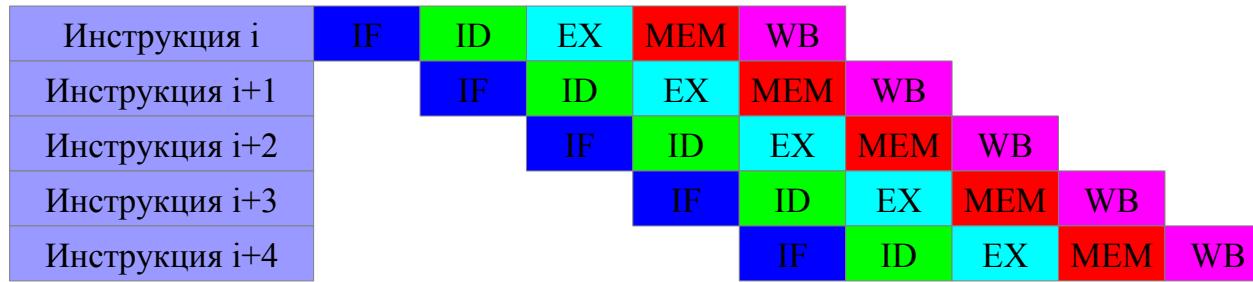


През 60-те години на миналия век с появата на *IBM System/360* започва използването на адресираната на ниво **байт (8 бита)** памет.

- ❖ На практика повечето компютри от тогава адресират и работят с набори от битове кратни на байт;
- ❖ Обработката на машинните думи в процесорите (8, 16, 32, 64 и т.н. бита) представлява парална обработка на ниво бит;

Instruction-Level Паралелизъм (ILP)

Инструкция №



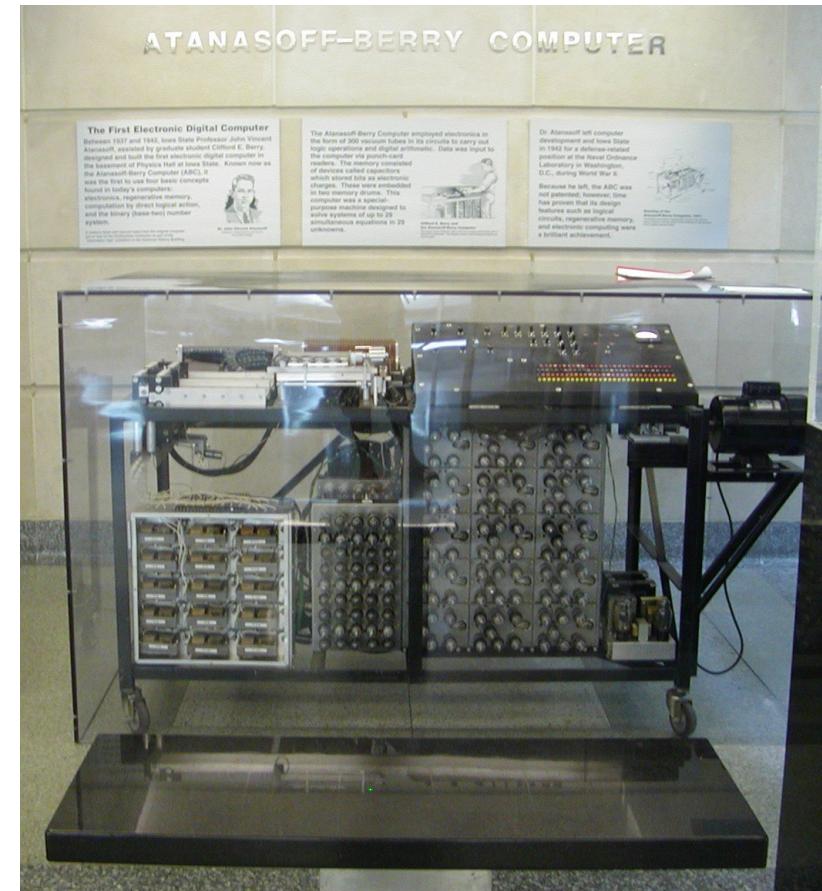
Pentium 4 с
34 стъпков pipeline

Скалатните и супер скаларните архитектури (за които говорихме в предишните лекции) са пример за паралелно изпълнение на ниво инструкция/инструкции. Обикновено решението е динамично...

- ❖ Instruction pipelining, Superscalar execution, Out-of-order execution, Register renaming, Speculative execution, Branch predicting, ...;
- ❖ ILP не трябва да се бърка с конкурентно изпълнение (изпълнение на последователности от инструкции в отделни нишки);



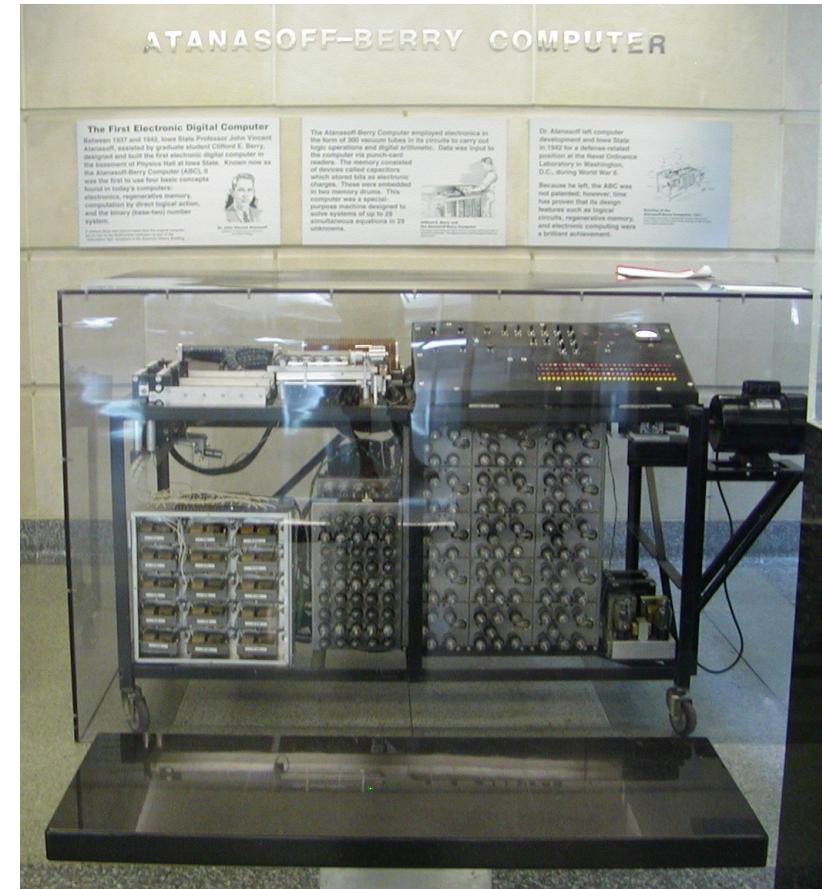
Първият компютър с паралелна обработка



Първият компютър с паралелна обработка

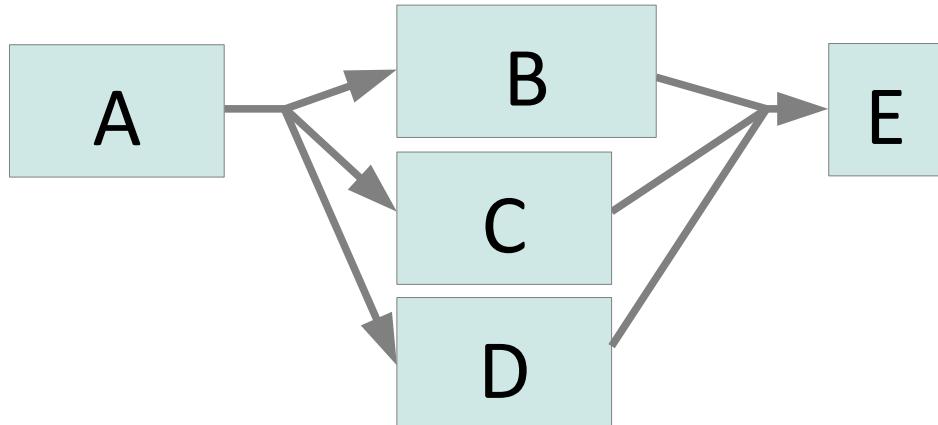
Компютърът ABC

- ❖ Не е нито програмираме, нито Turing complete и е можел е да пресмята само системи линейни уравнения;
 - ❖ Бинарна аритметика;
 - ❖ Електронни елементи;
- ❖ Междинна памет за резултатите, вход изход с хартиени карти, паралелна обработка.

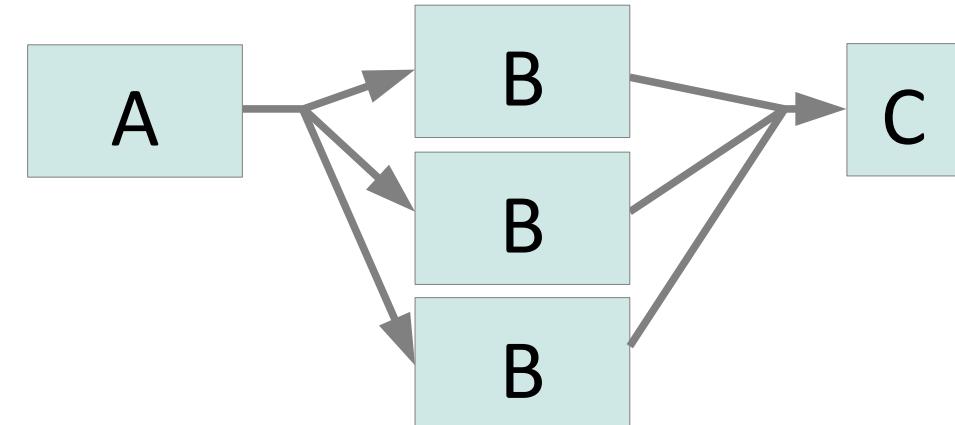


Функционален и Даннов Паралелизъм

- ❖ Функционален паралелизъм (Functional parallelism) – Всеки процесор работи върху част от проблема (задачата) – разбиване на алгоритъма;
- ❖ Даннов паралелизъм (Data parallelism) – Всеки процесор извършва една и съща работа върху част от данните при решаването на проблема – разбиване на данните;



Функционален



Даннов

Задачи

фибри, нишки, процеси



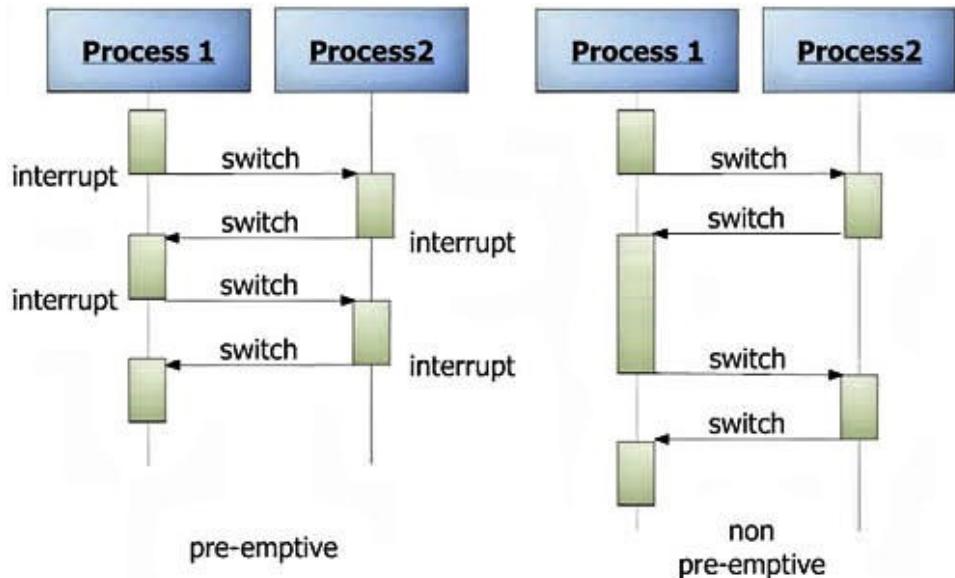
Задачи

Задача в паралелното програмиране се нарича код или последователност от инструкции, който се изпълняват конкурентно и понякога кооперативно с друг код.

Видове:

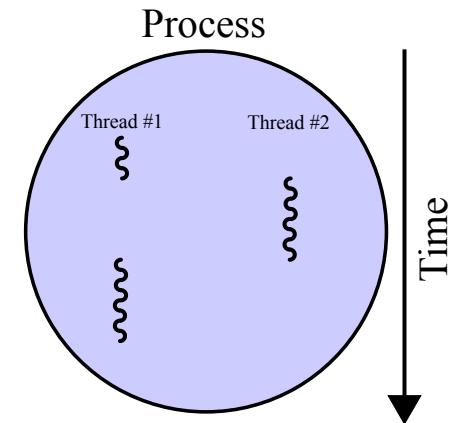
Фибри (fibers, coroutines), Нишки (Threads), Процеси (Processes);

Фибри



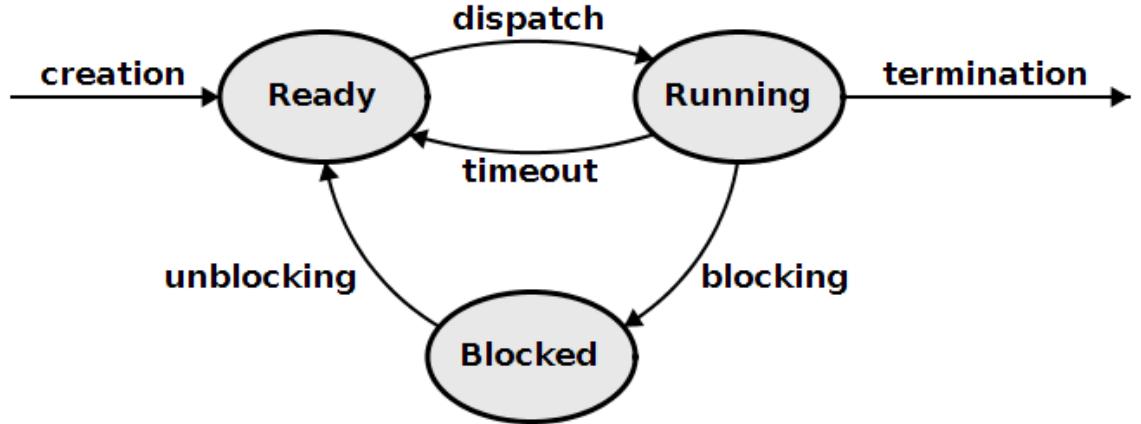
- ❖ Фибри (Fibers) – Леки нишки; Споделят адресното си пространство; За разлика от нишките фибрите използват кооперативна многозадачност, а не „превантивна“ многозадачност (preemptive multitasking);
Coroutine е синоним;
Могат да има състояние или да нямат;

Нишки



- ❖ Нишки (Threads) – Нишките са съвместно (конкурентно) изпълняващи се парчета код. Често това изпълнение се поддържа от ОС. Нишките споделят ресурси (адресно пространство, един или повече процесори/ядра и др.). Най-често са под управлението на „превантивна“ многозадачност (preemptive multitasking).
Използват се или Времева многозадачност (Temporal) или Едновременна многозадачност (Simultaneous multithreading);

Процеси



- ❖ Процеси (Proceses) – Процесите са отделни програми в ОС. Имат отделно (и защитено от ОС) адресно пространство. Най-често са под управлението на „превантивна“ многозадачност (preemptive multitasking). Може да се състоят от множество фибри и/или нишки.;

Задачи

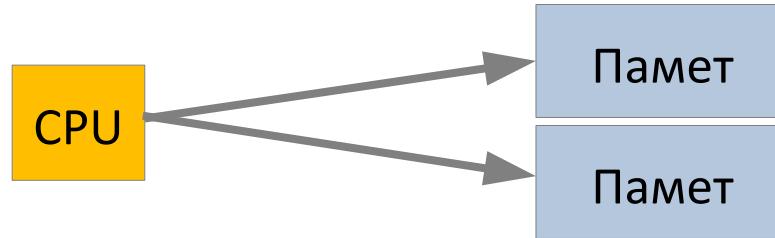
В някои езици (Ada, D и др.) задачите са вградени и са част от езика. В други езици се използват библиотеки (много често те са част от framework) от функции или класове (C++, C#, Java).

Даннов

MLP, Векторизация, ...



Memory-Level Паралелизъм (MLP)



Memory-level parallelism (MLP) е термин в компютъните архитектури, който е свързан с възможността да имаме повече от една чакащи операции с паметта в даден момент.

- ❖ Това е характерно за ILP процесорите, но може да се срещне и при някои форми на prefetching или hardware scouting;
- ❖ Също се отнася и до достъп кеша и др.;

Векторизация

- ❖ Векторизация (vectorization) е процес на преобразуване на програма изпълняваща една операция за всеки такт от време (в една нишка) към програма изпълняваща няколко операции едновременно в един такт (в една нишка);
- ❖ Векторизацията е частен случай на паралелизацията;
- ❖ Обикновено се използват SIMD инструкциите на съвременните процесори;
- ❖ (MMX, SSE, AVX, Altivec, NEON, ...);

Как се прилага

- ❖ Ръчно;
- ❖ Ръчно с използване на *intrinsic* функции;
- ❖ Автоматично от компилатора;
Обикновено компилатора трябва да бъде „подпомогнат“ от нас.
- ❖ Автоматично от процесора;
Като част от скаларните и супер-скаларните архитектури.

Пример 1

```
for (i = 0; i < 1024; i++)
    C[i] = A[i] + B[i];
```

Векторизация на цикли

```
for (i = 0; i < 1024; i+=4)
    (C[i], C[i+1], C[i+2], C[i+3]) =
        (A[i], A[i+1], A[i+2], A[i+3]) +
        (B[i], B[i+1], B[i+2], B[i+3]);
```

псевдокод

Частично развиване на цикъл.

Може ли границите на цикъла да не са кратни на 4?



Пример 2

```
for (i = 0; i < MAX; i++)
{
    C[i].x = A[i].x + B[i].x;
    C[i].y = A[i].y + B[i].y;
    C[i].z = A[i].z + B[i].z;
}
```

Векторизация в блок

```
for (i = 0; i < MAX; i++)
{
    (C[i].x, C[i].y, C[i].z) =
        (A[i].x, A[i].y, A[i].z) +
        (B[i].x, B[i].y, B[i].z);
}
```

псевдокод

Ако дължината на вектора е 4, то в примера има проблем с непълното използване на SIMD възможностите



Пример 3

```
for (i = 0; i < 1024; i++)
{
    if (A[i] > 0)
        C[i] = B[i];
    else
        D[i] = D[i-1];
}
```

Векторизация на
разклонени алгоритми

```
for (i = 0; i < 1024; i++)
{
    P = A[i] > 0;
    NP = !P;
    C[i] = B[i];    (P)    // изпълнява се само ако P е истина
    D[i] = D[i-1]; (NP)    // изпълнява се само ако NP е истина
}
```

псевдокод



Пример 3

```
for (i = 0; i < 1024; i++)
{
    if (A[i] > 0)
        C[i] = B[i];
    else
        D[i] = D[i-1];
}
```

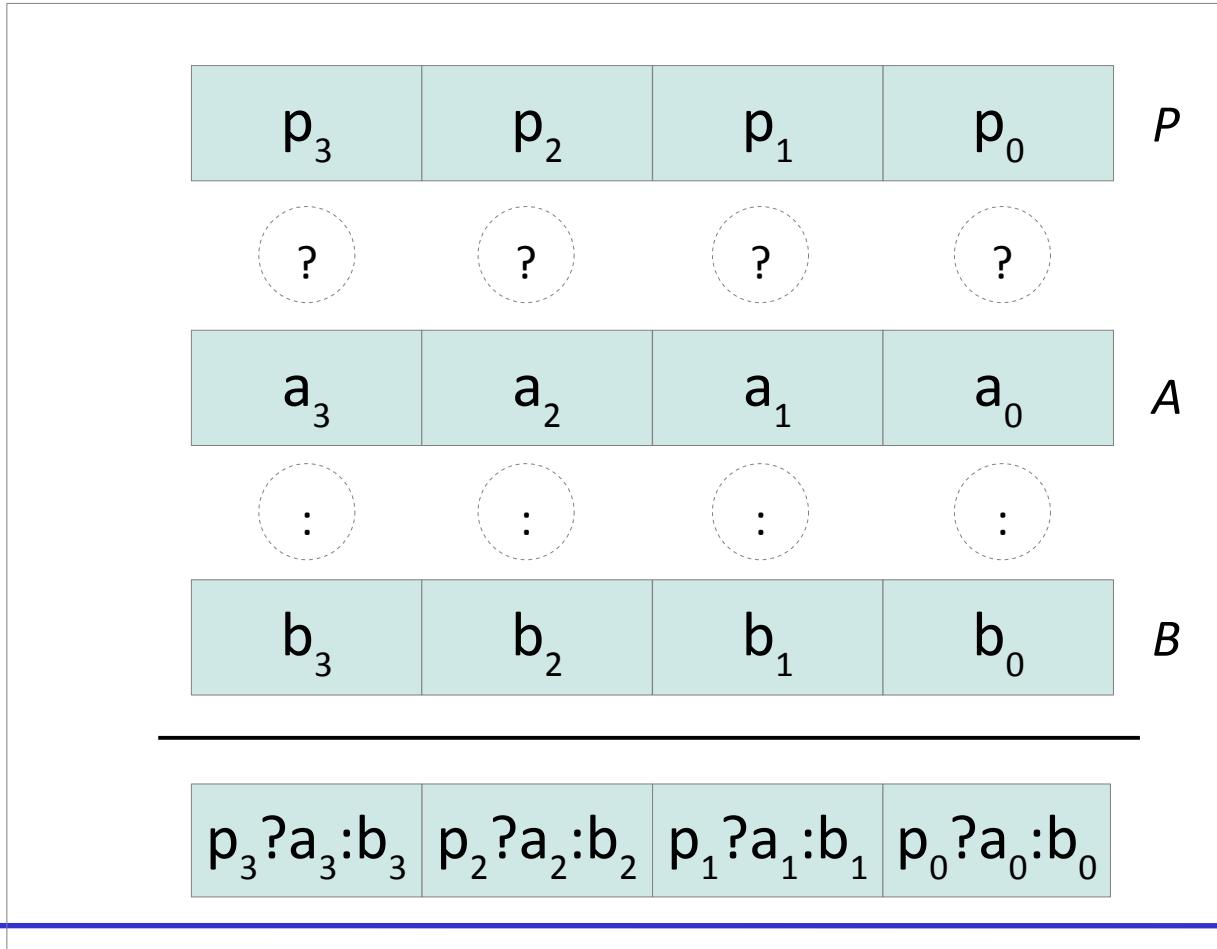
Векторизация на
разклонени алгоритми

```
for (i = 0; i < 1024; i+=4) {
    vP = (A[i], A[i+1], A[i+2], A[i+3]) > (0, 0, 0, 0);
    vNP = !vP; // векторно т.е. покомпонентно отрицание
    (C[i], C[i+1], C[i+2], C[i+3]) =
        vsel(vP, (B[i], B[i+1], B[i+2], B[i+3]), (C[i], C[i+1], C[i+2], C[i+3]));
    if (vNP[4]) D[i+3] = D[i+2]; if (vNP[3]) D[i+2] = D[i+1];
    if (vNP[2]) D[i+1] = D[i]; if (vNP[1]) D[i] = D[i-1];
}
```

псевдокод



Пример 3 – Векторна ϕ -я vsel(P, A, B)



Векторизация на цикли – общ случай

Всеки цикъл се развива до определено ниво (зависи от границите т.е. дали са известни по време на компилация и колко са големи).

Циклите се разделят на 4 части (или по-малко):

❖ Пред цикъл

Операции независими от цикъла (инварианти). Обикновено се зареждат (инвариантни) данни във векторни регистри и други

❖ Цикъл (Цикли)

Векторизирани вариант(i) на цикъла (циклите)

❖ След цикъл

*Получаване на резултати и допълнителна инвариантна обработка
Индукции, редукции и други*

❖ Опашка на цикъл

Реализация на невекторизиран вариант на цикъла за оставащите итерации, които по някакви причини не са попаднали в основния цикъл (например броя итерации не е кратен на големината на векторите или размера се определя в runtime)

Не всичко може да се векторизира

```
for (i = 0; i < 3; i++)  
{  
    C[i] = A[i] + B[i];  
}
```

OK.

Цикълът може да бъде развит и/или векторизиран напълно

```
for (i = 0; i < 1024; i++)  
{  
    C[i] = A[i] + B[i];  
}
```

OK.

Цикълът може да бъде векторизиран

```
for (i = 0; i < 1024; i++)  
{  
    D[i] = E[i] - A[i-1];  
    A[i] = B[i] + C[i];  
}
```

Не може!

Използва се стойност преди да бъде пресметната

Масив от структури или няколко масива?

?	Z _i	Y _i	X _i	v1
	+	+	+	+

?	Z _i	Y _i	X _i	v2
	+	+	+	+

X _{i+3}	X _{i+2}	X _{i+1}	X _i	v1x
	+	+	+	+

X _{i+3}	X _{i+2}	X _{i+1}	X _i	v2x
	+	+	+	+

```
struct { float x, y, z; } vec;  
  
vec[100] v1, v2;  
  
for (i = 0; i < 100; i++) {  
    v1[i].x += v2[i].x;  
    v1[i].y += v2[i].y;  
    v1[i].z += v2[i].z;  
}
```

ИЛИ

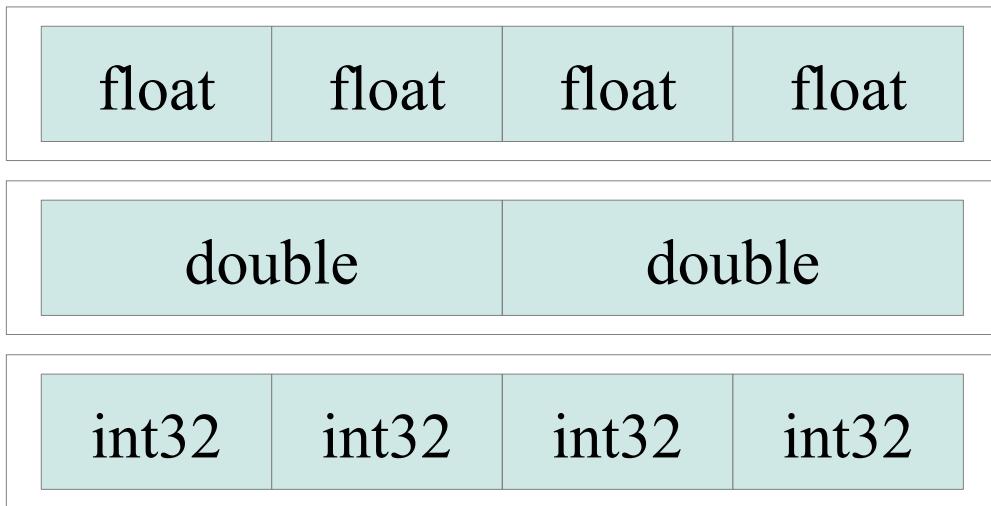
```
float[100] v1x, v2x;  
float[100] v1y, v2y;  
float[100] v1z, v2z;  
  
for (i = 0; i < 100; i++)  
    v1x[i] += v2x[i];  
for (i = 0; i < 100; i++)  
    v1y[i] += v2y[i];  
for (i = 0; i < 100; i++)  
    v1z[i] += v2z[i];
```



Типове данни

❖ Ако типа е еднакъв

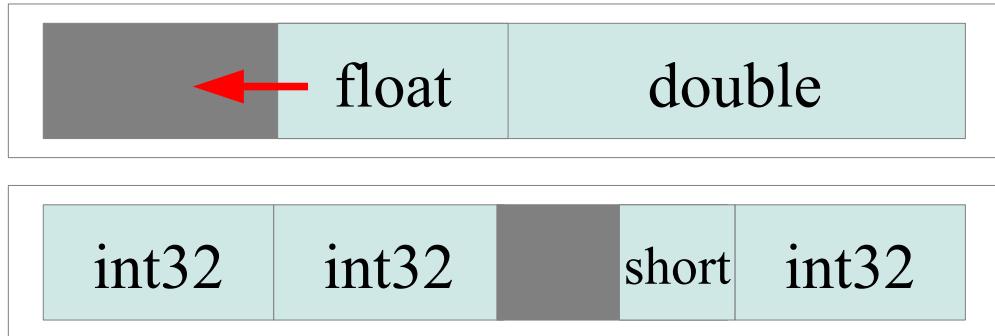
Например 4 float числа се събират точно в един SSE регистър



Типове данни

❖ Ако типовете са различни

Имаме загуба на производителност. Трябва да се внимава с Align на структурите
За някои типове може да се налага разширяване до по-голям тип (float->double)



Разпаралеляване на Цикли

Разпаралеляване на цикли

Много често циклите са линеаризирана версия на паралелен алгоритъм. Векторизацията на цикли е един от възможните подходи, но тя може да не е достатъчна или да не е подходяща за конкретния случай.

В зависимост от данновите и другите зависимости в цикъла, той може да е напълно неподходящ за разпаралеляване, да е подходящ за векторизация или друг вид даннов или задачен паралелизъм.



Многозадачность

*Temporal (TMT), Simultaneous (SMT), Speculative (SpMT),
Preemptive, Cooperative,
Clustered Multi-Thread (CMT)...*

Времева многозадачност (Temporal – TMT)

Времева многозадачност е една от двете основни форми на многозадачност, които могат да бъдат реализирани хардуерно.

- ❖ Броя на едновременно изпълняващите се задачи е $N=1$;
- ❖ Антипод на този вид е едновременната многозадачност ($N>1$);
- ❖ Два са основните под вида ТМТ:
 - ❖ Груб/Едрозърнест (Coarse-grained) – процесорът има само една линия за изпълнение и процесора трябва ефективно да превключва между отделните контексти. Това може да става на базата на време, изминали цикли, пропуски в кеша и др.;
 - ❖ Фин/Дребнозърнест (Fine-grained или Interleaved) – процесорът може да има повече конвейери за изпълнение;

Едновременна многозадач. (Simultaneous – SMT)

Едновременна многозадачност е техника за подобряване на общата ефективност на суперскаларните процесори с хардуерна многозадачност. SMT позволява множество независими нишки на изпълнение за по-добро използване на ресурсите.

- ❖ Броя на едновременно изпълняващите се задачи/нишки на едно ядро е $N > 1$;
- ❖ Позволено е да се изпълняват много задачи с различни права, памет, В/И права, както и на различно ниво на защита (rings);
- ❖ Този подход е подобен на preemptive multitasking, но е реализиран хардуерно на ниво нишки в модерните супер скаларни процесори;



Кълстерирана многозадачд. (Clustered multithr. – CMT)

При **Кълстерирана многозадачност** (CMT) някои части на процесора се споделят между две нишки, а някои части са уникални за всяка нишка.

- ❖ CMT е по-прост вариант на SMT;
- ❖ Целта е да се оползовтвоти неизползвания хардуер при изпълнение на повече задачи едновременно;

Спекулативна многозадачност. (Speculative – SpMT)

Спекулативна многозадачност, известна още като Thread Level Speculation (TLS), е техника за спекулативно изпълнение на част от кода, който се очаква да бъде изпълнен по-късно паралелно с нормалното изпълнение на отделна независима нишка.

- ❖ Прави предположение за входните променливи;
- ❖ Изпълнява (“спекулативно”) кода, така все едно предположението е вярно;
- ❖ Ако предположението в последствие се окаже неуспешно, то резултата от изпълнението се игнорира или се изчислява/изпълнява наново;

“Превантивна” многозад. (Preemptive multithr.)

При **“превантивната” многозадачност** се извършва временно прекъсване на задача, изпълнявана от компютърна система, без да се изисква нейното сътрудничество. Това се прави с намерението да се възобнови задачата в по-късен момент.

- ❖ Обикновено това се извършва от ОС т.е. от нейния планировчик на задачи (preemptive scheduler);
- ❖ За целта се използва превключване на контекста от текущата задача към контекста на друга;
- ❖ За разлика от кооперативната многозадачност не се изисква задачата изрично да освободи ресурса (CPU) с „yield”.



Кооперативна многозадач. (Cooperative multithr.)

Кооперативната многозадачност (също известна като non-preemptive) е вид многозадачност, при която ОС не инициира (принудително) превключване на задачите. Вместо това:

- ❖ Процесите (задачите) доброволно освобождават ресурса и връщат контрола на ОС, когато са свободни или са блокирани поради вътрешни причини;
- ❖ Това позволява на другите процеси да се изпълняват конкурентно;
- ❖ Това може да доведе до някои проблеми вкл. да „забие“ цялата ОС;



Въпроси?

apenev@uni-plovdiv.bg



Паралелно Програмиране

Теоретични аспекти на паралелните алгоритми.
Анализ на паралелни алгоритми.
Зависимости на данните, структурата и контрола.

PRAM Model

PRAM модел

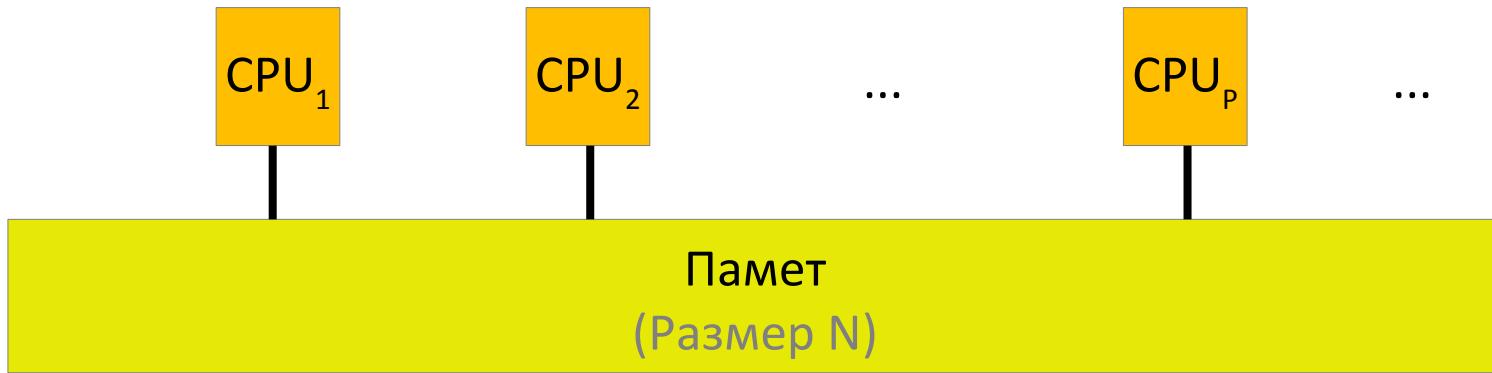
- ❖ Parallel Random-Access Machine (PRAM) е абстрактна машина със Споделена памет (Shared memory);
- ❖ Моделът пренебрегва практически проблеми като синхронизация и комуникация (по същия начин както RAM модела пренебрегва разликите във времената за достъп до кеша и основната памет);
- ❖ Моделът “осигурява” 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 създават проблеми и се класифицират като:

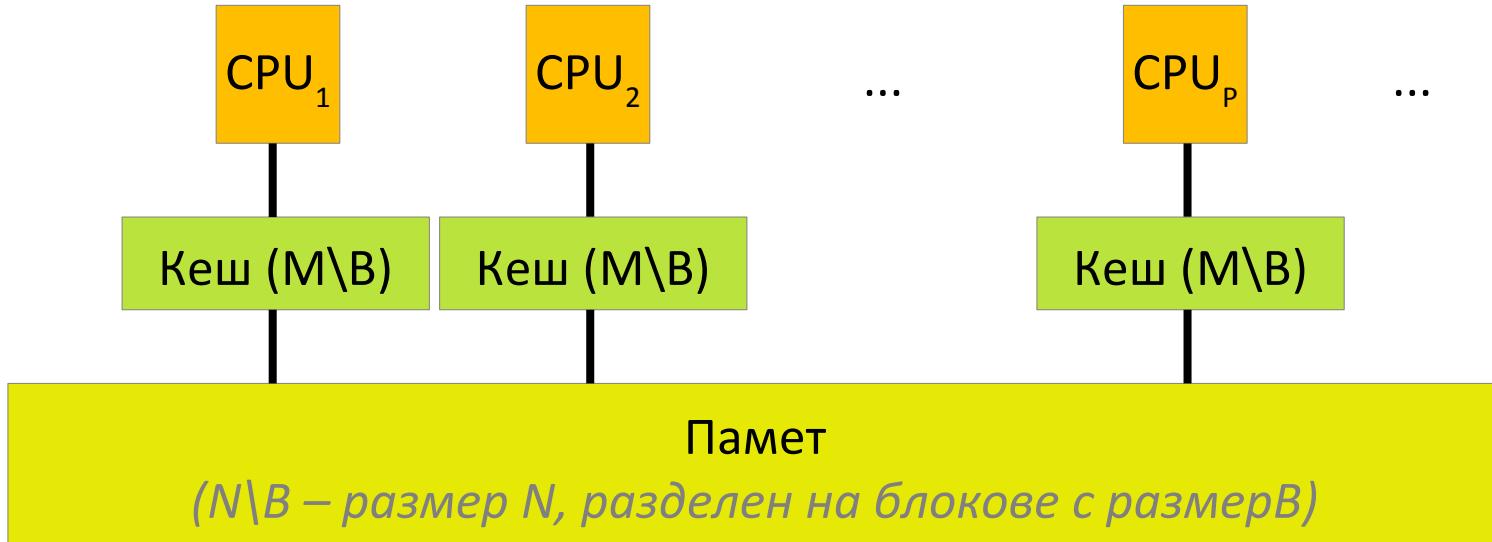
- ❖ Общи (Common) – всички процесори записват една и съща стойност; (ако не е така е недопустимо/проблем)
- ❖ Произволни (Arbitrary) – само един произволен опит е успешен, а останалите се отхвърлят;
- ❖ Приоритетен (Priority) – ранга на процесора показва кой ще запише;
- ❖ Друг вид – например при различните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

РЕМ 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

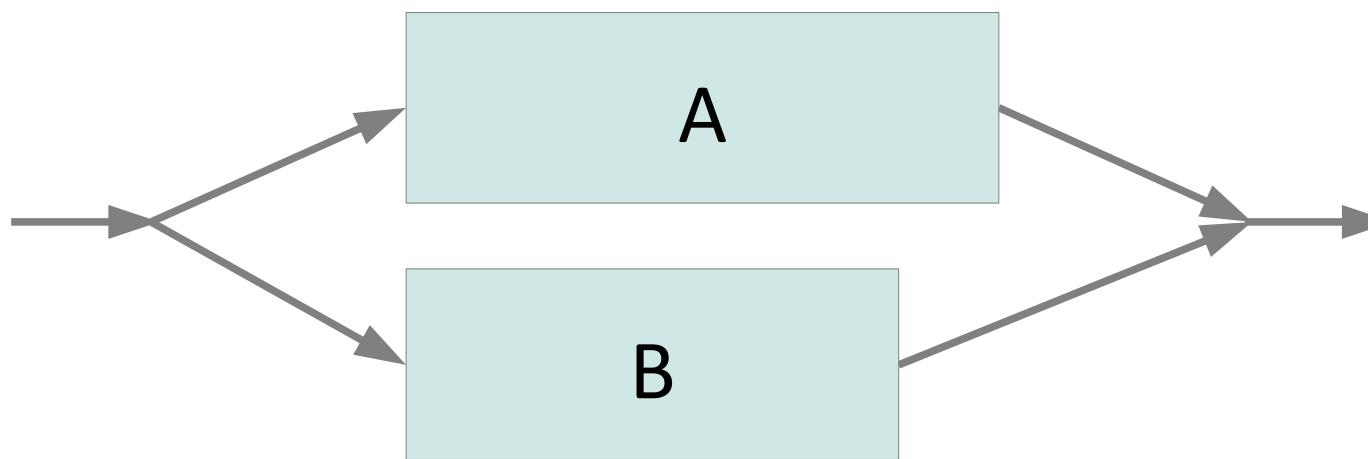
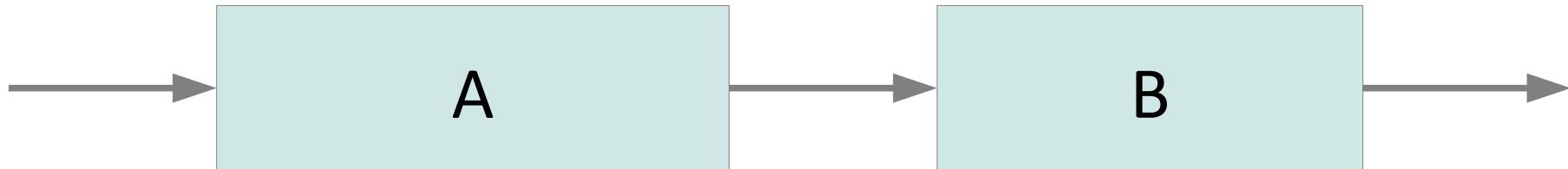


Разпаралеляване

Какво е разпаралеляване (паралелизация)?

- ❖ 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)?

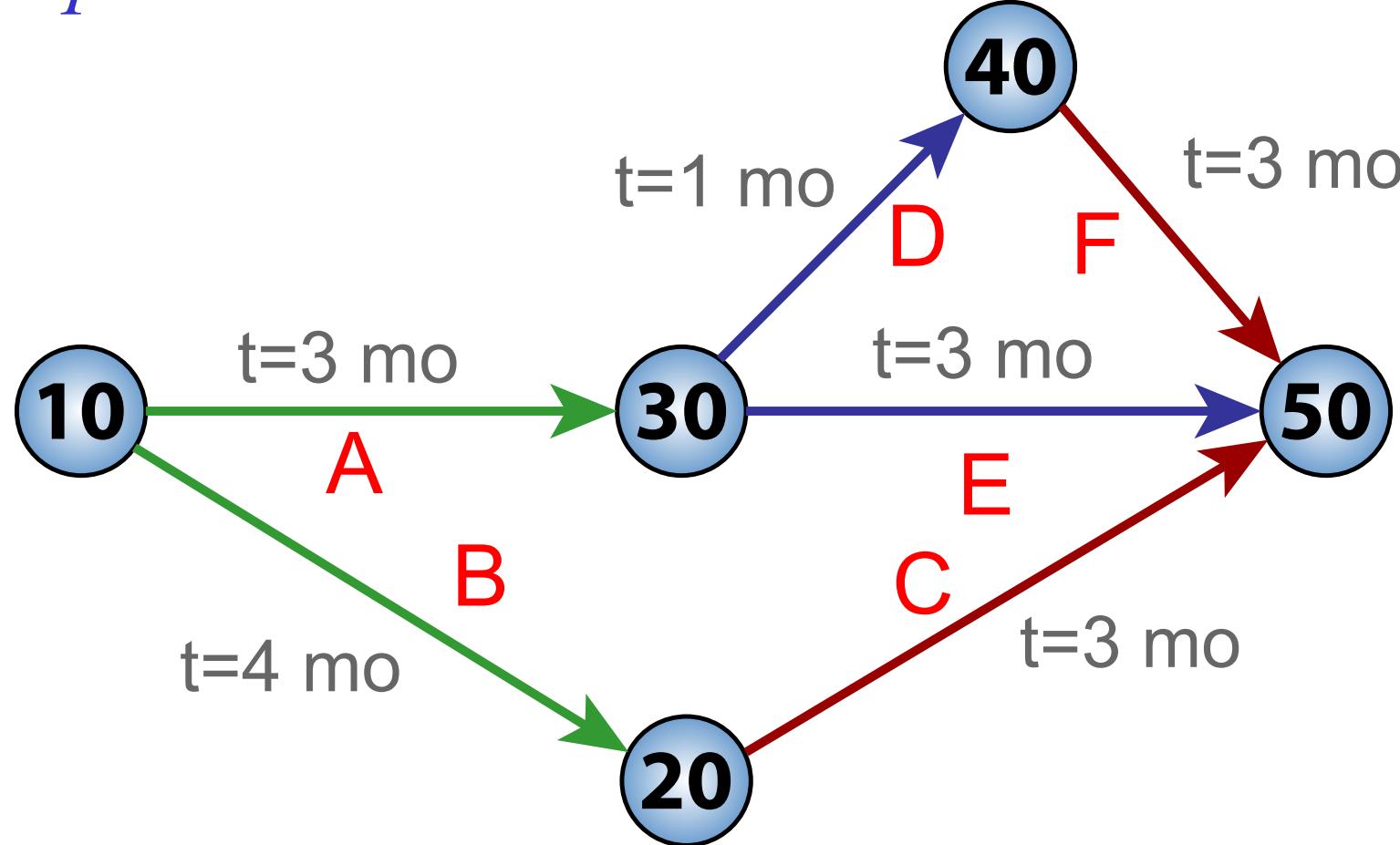


Как се реализира?

- ❖ 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

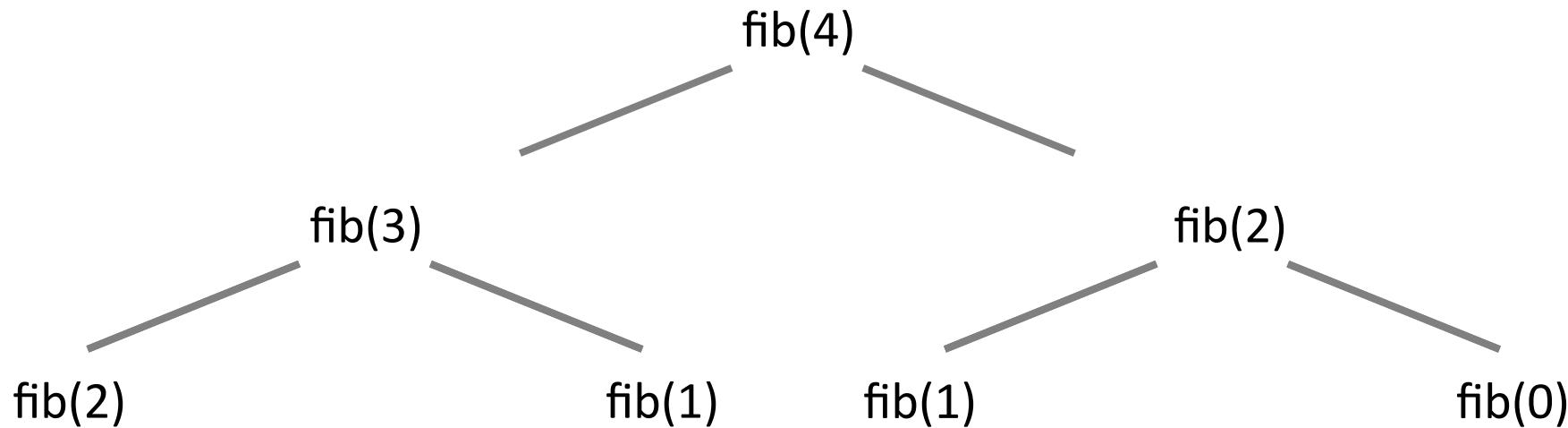
```
int fib(int n) {  
    if (n < 2) return n;  
  
    int n1, n2;  
  
    #pragma omp task shared(n1)  
    n1 = fib(n - 1);  
    #pragma omp task shared(n2)  
    n2 = fib(n - 2);  
  
    #pragma omp taskwait  
  
    return n1 + n2;  
}
```

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 Фиbonачи (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_∞ – 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

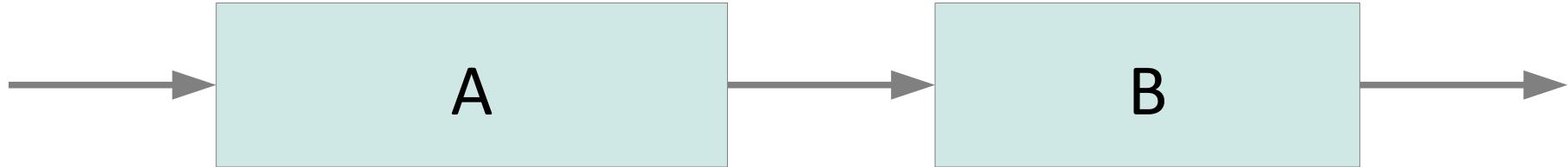
- ❖ Закон за Работата (work law):

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

- ❖ Закон за Периода (span law):

$$T_P \geq T_\infty$$

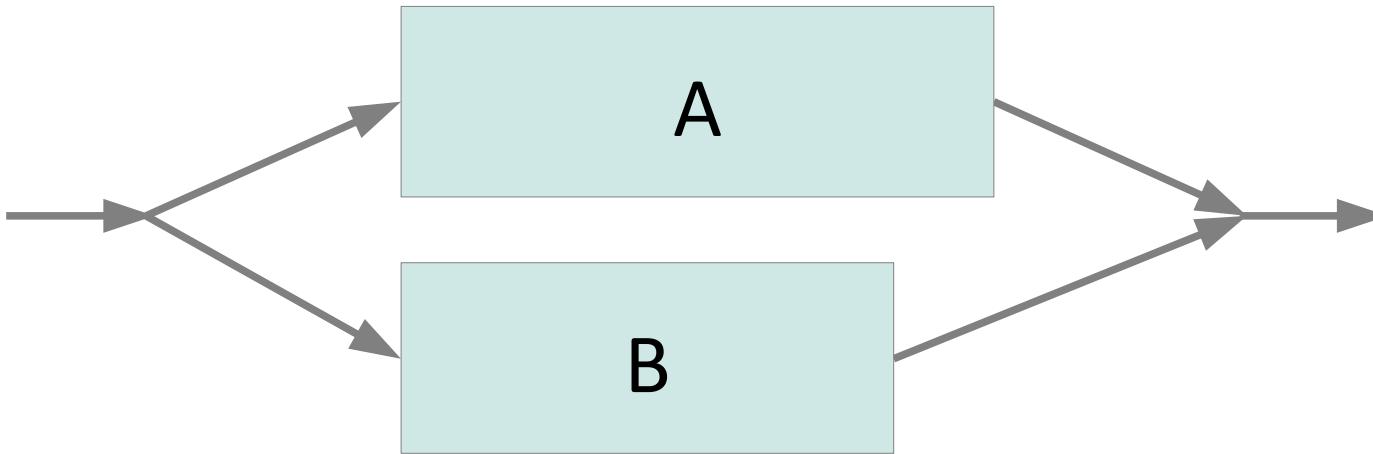
Времена на паралелен алгоритъм



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

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

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 $T_1/T_P = k * P$
- ❖ 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...

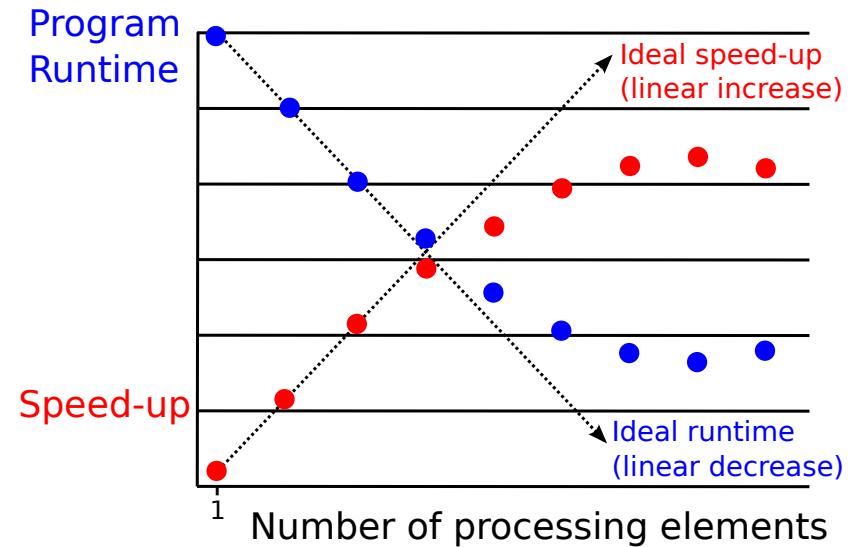
Скалируемост (Scalability)

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 1000×1000 имаме

Паралелизъм $\approx 10^6$

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

- ❖ За един нормален RayTracer (при сцена 1000×1000 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

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

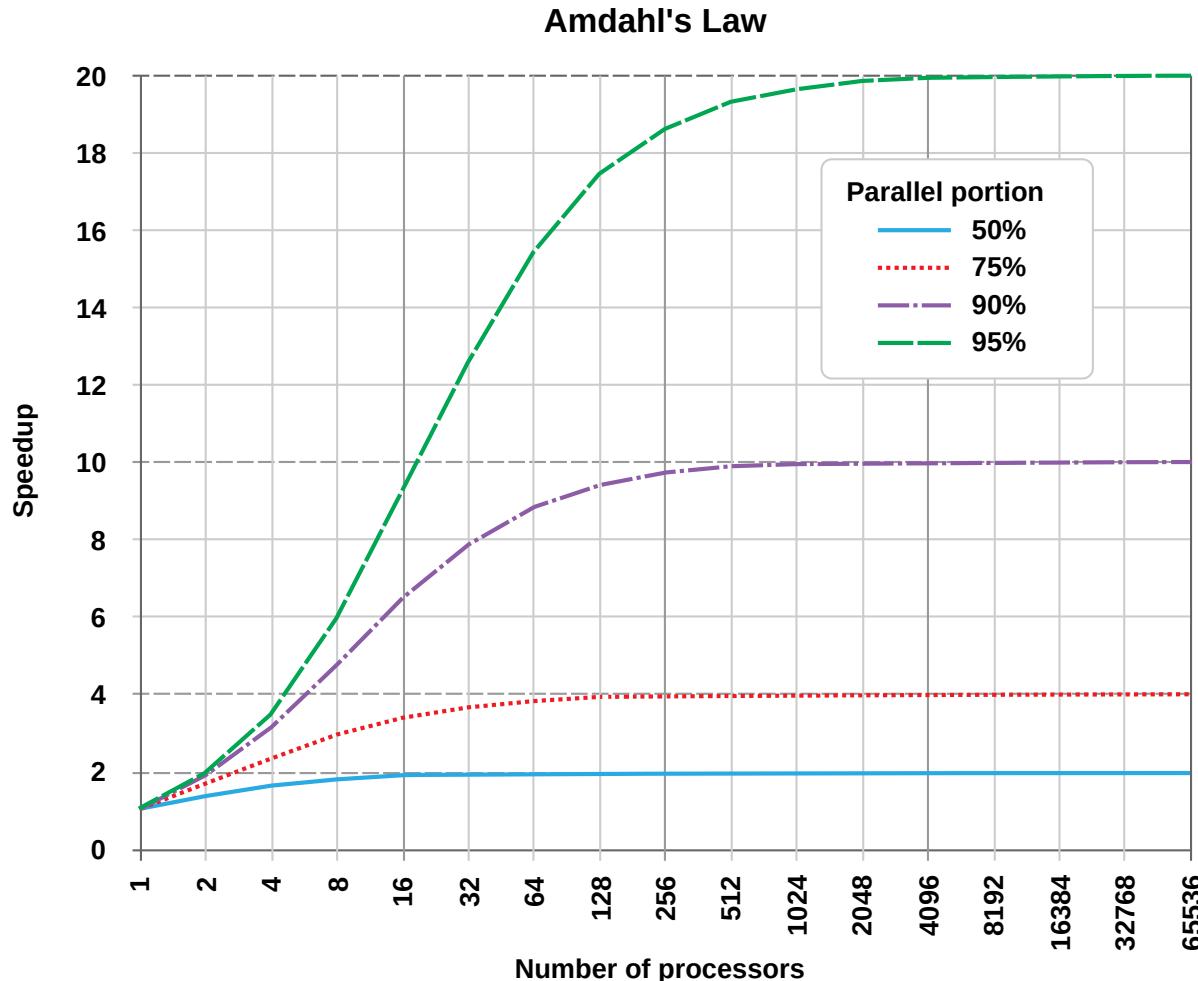
- ❖ Ако този коефициент е по-малък от 1, то перфектното линейно ускорение е невъзможно.

Закон на Амдала

Amdahl



Закон на Амдала /Amdahl/ (1967)



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

$S_{latency}$ е теоретичното ускорение;

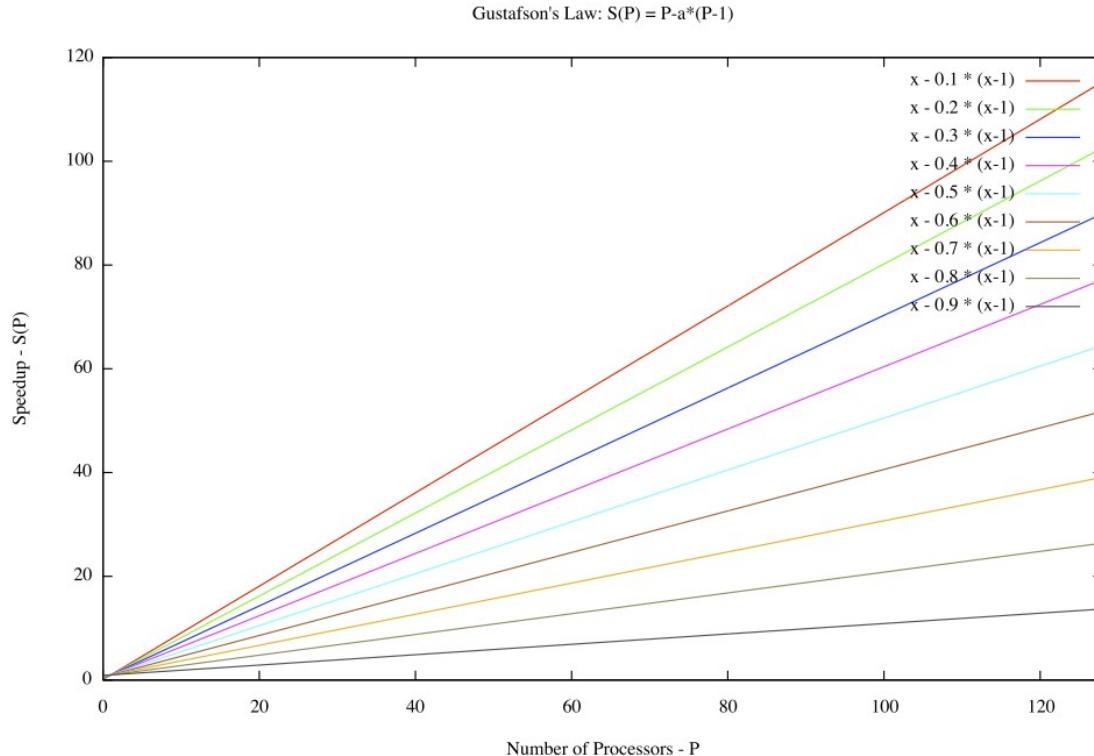
S е ускорението на частта от задачата, която се възползва от тези паралелни ресурси;

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 подобрението.

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 ψ на p процесора, където $p > 1$, експериментално определената сериийна 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 сериината фракция (не паралелната част);

T_p е времето на паралелната част изпълнена на един процесор;

p е броя на процесорите;

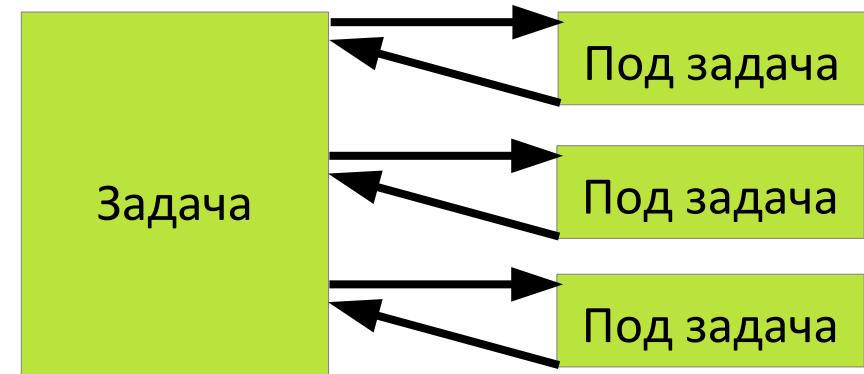
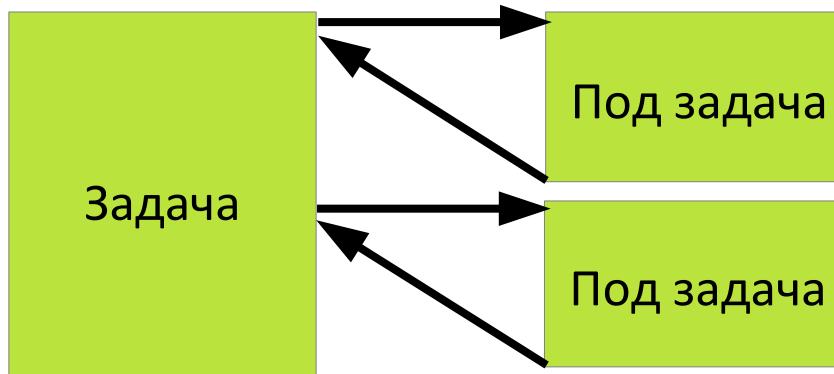
Детайлност/Прецизност *(Granularity)*



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;



Въпроси?

apenev@uni-plovdiv.bg



Паралелно Програмиране

Проблеми при паралелните алгоритми.

Задача за „Вечеряящите философи“.

Мъртва хватка, жива хватка, трудна скалируемост,
глад за ресурси, съперничество и др.

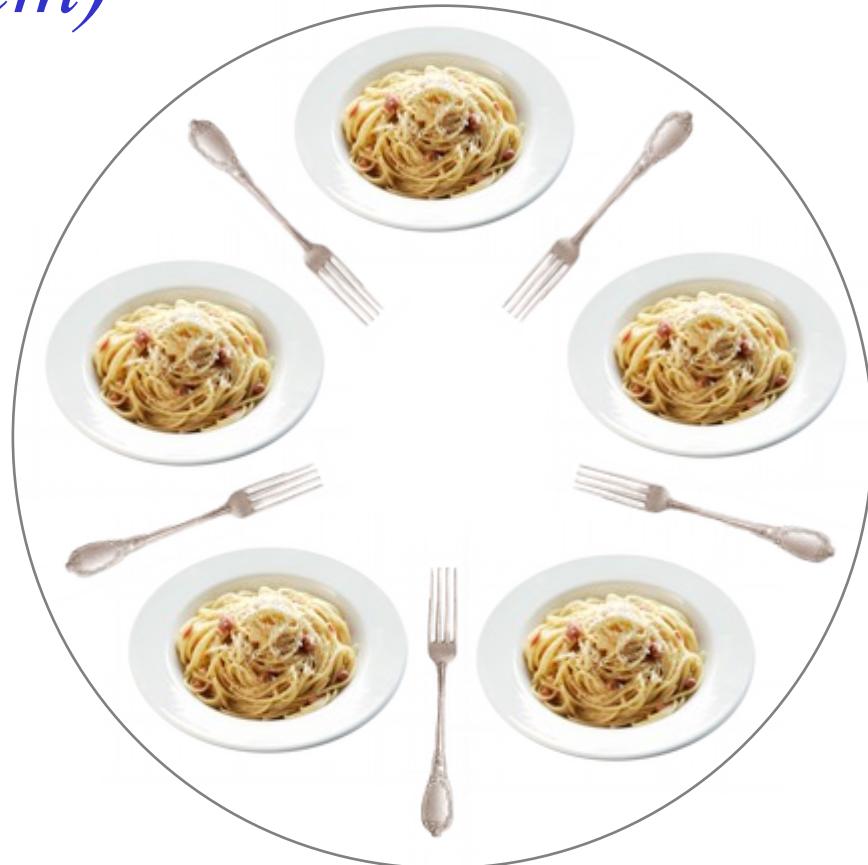
Producer-Consumer.

доц. д-р Александър Пенев

Задача за Вечерящите Философи

Вечерящите философи (Concurrency, Паралелност)

- ❖ Пет мълчаливи философи седят на кръгла маса с чинии със спагети;
- ❖ Вилици се поставят между всяка двойка съседни философи;
- ❖ Всеки философ трябва последователно да мисли и да се храни;



Вечерящите философи (Concurrency, Паралелност)

- ❖ Философът обаче може да яде спагети само когато имат и лява, и дясна вилица;
- ❖ Всяка вилица може да се държи само от един философ и така философът може да използва вилицата само ако не се използва от друг философ;



Вечерящите философи (Concurrency, Паралелност)

- ❖ След като отделен философ приключи с храненето, те трябва да сложат и двете вилици, така че вилиците да станат достъпни за другите;
- ❖ Може да вземе вилицата само отлясно или тази отляво, когато станат достъпни и те не могат да започнат да ядат, преди да получат и двете вилици;



Проблем

- ❖ Проблемът е как да се проектира дисциплина на поведение (паралелен алгоритъм), така че никой философ да не гладува; т.е. всеки може завинаги да продължи да редува хранене и мислене (никой философ не може да знае кога другите могат да искат да ядат или да мислят).



Вечерящите философи

- ❖ Взаимното изключване (mutual exclusion) е основната идея на проблема;
- ❖ Провалите, които тези философи могат да изпитат, са аналогични на трудностите, които възникват при реалното компютърно програмиране, когато много програми се нуждаят от изключителен достъп до споделени ресурси;

Вечерящите философи

- ❖ Първоначалните проблеми на Dijkstra са свързани с външни устройства като касетни устройства. Въпреки това, трудностите, обяснени от проблема с философите на трапезарията, възникват много по-често, когато множество процеси имат достъп до набори от данни, които се актуализират;
- ❖ Системи като ядрото на ОС, използват хиляди заключвания и синхринизации, които изискват стриктно следване на протоколите за да се избегнат проблеми като мъртви хватки и др.

Възможни Решения

- ❖ Въвеждане на йерархия на ресурсите – между ресурсите се въвежда частична наредба (номерираме ги), след което когато даден философ взема ресурс (от вилиците номерирани от 1 до 5), то той трябва да го прави в строго нарастващ ред по реда на ресурсите. Макар и решение то не винаги води до ефективни системи;
- ❖ Арбитриране – за да се гарантира, че философите ще вземат винаги две вилици, се въвежда арбитър (сервитьор), който разрешава ползването на ресурсите само на един философ в даден момент. Връщането става във всеки момент;
- ❖ Chandy-Misra;

Алгоритъм на Декер

Алгоритъм на Декер (Dekker)

- ❖ Алгоритъмът на Декер е първото известно решение на проблема с взаимното изключване в паралелното програмиране;
- ❖ Той позволява на две нишки да споделят ресурс за еднократна употреба без конфликт, като се използва само споделена памет за комуникация;
- ❖ При реализация в модерните компютри трябва да се използва бариера (memory barrier), да се внимава за оптимизацията;
- ❖ При езиците предлагащи “volatile” е модifikатора да бъде използван пред променливите;

Алгоритъм на Декер (Dekker)

```
vars: wants_to_enter : array of 2 booleans ← {false, false}
      turn : integer ← 0    // or 1

P0: wants_to_enter[0] ← true
while wants_to_enter[1] {
  if turn ≠ 0 {
    wants_to_enter[0] ← false
    while turn ≠ 0 { // wait }
    wants_to_enter[0] ← true
  }
}

// critical section ...
turn ← 1
wants_to_enter[0] ← false

P1: wants_to_enter[1] ← true
while wants_to_enter[0] {
  if turn ≠ 1 {
    wants_to_enter[1] ← false
    while turn ≠ 1 { // wait }
    wants_to_enter[1] ← true
  }
}

// critical section ...
turn ← 0
wants_to_enter[1] ← false
```



Задача за Пушачите

Задача за Пукачите



The smoker
(*The X-Files*)



Thin Man
(*Charlie's Angels*)

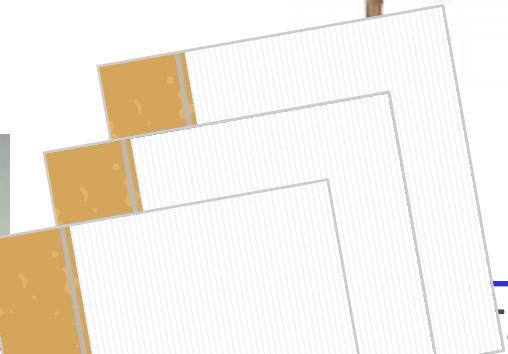


Catherine Tramell
(*Basic Instinct*)

Задача за Пушачите

- ❖ Да приемем, че една цигара изисква три съставки за приготвяне и пущене: тютюн, хартия и кибрит;
- ❖ Около масата има трима пушачи, всеки от които има безкрайна доставка на една от трите съставки – един пушач има безкраен запас тютюн, друг има хартия, а третият има кибрит;
- ❖ Има и агент за непушачи, който дава възможност на пушачите да си правят цигари, като произволно (недетерминирано) избира два от консумативите, които да поставят на масата;
- ❖ Пушачът, който има третата съставка, трябва да вземе двете съставки от масата, като ги използва (заедно със собствената си съставка), за да си направи цигара, която пуши известно време.

Задача за Пукачите



Реализация (псевдокод)

```
def tobacco_smoker():
    repeat:
        paper.wait()
        matches.wait()
        smoke()
        tobacco_smoker_done.signal()
```

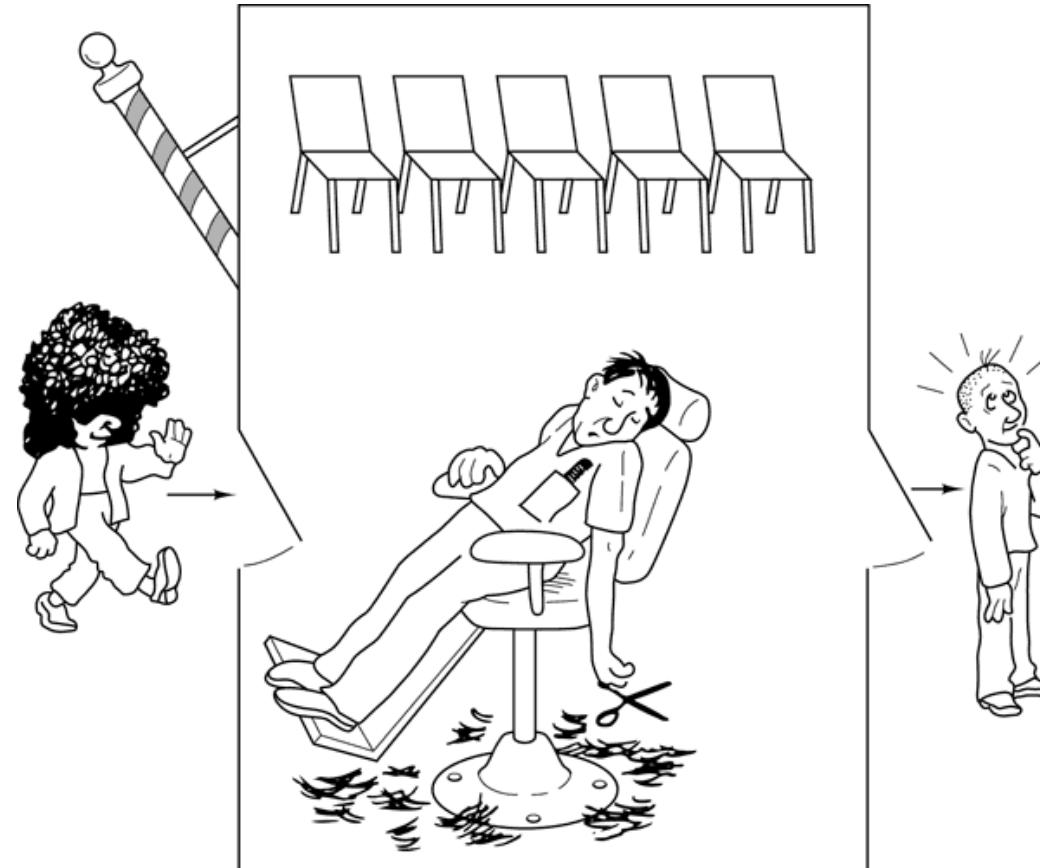
Другите две реализации за пушачите са аналогични.

Проблем и Решение

- ❖ Когато агентът постави примерно тютюн и хартия на маса, може притежаващия тютюн пушач да вземе хартията, което води до мъртва хватка;
- ❖ Решението е да се използват три “семафора” за да описват поставените на маса съставки, както и три “семафора” за всеки пушач, който информира агента, че съответния пушат е приключи с пущенето;

Задача за Спящия Бърснар

Задача за Спящия бърснар



Задача за Спящия бръснар

- ❖ Нека да имаме бръснарница с един бръснар и един стол за подстригване;
- ❖ Нека имаме чакалня с N на брой стола;
- ❖ Когато няма клиенти бръснарят сядат на стола и спят;
- ❖ Когато дойде клиент той вижда дали има друг клиент на стола:
 - ❖ Ако да – сядат в чакалнята или си тръгват (ако няма места);
 - ❖ Ако не – влиза, събържда бръснаря и сядат на стола;
- ❖ Когато бръснарят свърши с подстригването, той изпраща клиента и гледа дали има клиенти в чакалнята:
 - ❖ Ако няма – сядат на стола и заспиват;
 - ❖ Ако има – кани произволен от тях;

Особености и Проблеми при Паралелни Програми

Опасности и проблеми

- ❖ Мъртва хватка (Deadlock)
- ❖ Жива хватка (Livelock)
- ❖ Конвоиране (Convoying)
- ❖ Съперничество (Contention)
- ❖ Глад за ресурси (Starvation)
- ❖ Допълнително време (Overhead)
- ❖ Забавяне (Parallel slowdown)

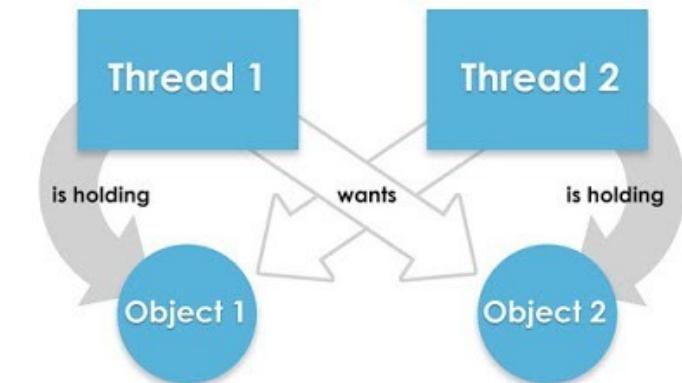
- ❖ Безпроблемни “Embarrassingly parallel”
- ❖ Детерминирани и недетерминирани алгоритми;

Мъртва хватка (Deadlock)

Мъртва хватка (Deadlock)

Мъртва хватка е проблем в паралелното програмиране, при който два или повече процеса взаимно се изчакват за достъп до общи ресурси, като всички процеси попадат в изчакващо състояние

Ситуацията прилича на тази, в която две коли се срещат на тесен път и нито едната може да мине, нито другата.



Решения при Мъртва хватка (Deadlock)

- ❖ Действия при откриване на мъртва хватка – спиране на процесите, спиране един по един до преодоляване на проблема;
- ❖ Алгоритъм на “Банкера”;
- ❖ Resource Preemption;
- ❖ Оптимистичен и писимистичен сценарии;

Жива хватка (*Livelock*)

Жива хватка (*Livelock*)

Жива хватка е проблем в паралелното програмиране, при който два или повече процеса взаимно се изчакват за достъп до общи ресурси, като за разлика от мъртвата хватка те не са в изчакващо състояние, а активно се опитват да получат ресурсите без да извършват никаква друга полезна работа.

Ситуацията прилича на тази, в която двама души се срещат лице в лице на коридора и всеки отстъпва път на другия, но после и двамата тръгват напред и пак се сблъскват и т.н.

Решения при Жива хватка (Livelock)

- ❖ По аналогия с мъртвата хватка за избягване на проблема;
- ❖ Живата хватка е дори по-опасна и трудна за откриване, защото процесите не “спят”, а извършват някаква “работка” – активно чакане, което отнема допълнителни ресурси и не извършва полезна работа;

Конвоиране (*Convoying*)

Конвоиране (Convoying)

Конвоиране е проблем в паралелното програмиране, при който за разлика от мъртвата хватка процесите не се блокират и работят, но непрекъснато се изчакват един друг, като въпреки че вършат и полезна работа, то тя е много малко, което води до драстично падане на производителността. Например може да се получи, че процесите чакат един от тях, а той изчаква нов квант време за да продължи работата си.

Решения при Конвоиране (Convoying)

- ❖ Анализ на работата на процесите;
- ❖ По-прецизна синхронизация;

Съперничество (*Contention*)

Съперничество (Contention)

Съперничество е проблем в паралелното програмиране, при който възниква “спор” за достъп до споделен ресурс, като памет с произволен достъп, дисково съхранение, кеш памет, вътрешни шини или външни мрежови устройства и др. Ресурс, за който има непрекъснати спорове, може да бъде определен като прекалено желан (необходим).

Разрешаване то на тези проблеми е една от основните функции на ОС.

Глад за ресурси (Starvation)

Глад за ресурси (Starvation)

Глад за ресурси е проблем в паралелното програмиране, при който имаме проблеми с общността на заключването:

Ред вместо таблица, клетка вместо ред, ...

Решения при Глад за ресурси (Starvation)

- ❖ Прецизиране на заключването;
- ❖ Анализ на алгоритъма и промени в него;

Допълнително време (Overhead)

Допълнително време (Overhead)

Допълнително време е проблем в паралелното програмиране, при който понякога времето за изпълнение на организационните дейности по паралелизация, критични секции и други е прекалено много в сравнение с реално извършваните действия на алгоритъма.

Такова време винаги съществува, но проблема е когато то е прекалено голям процент от общото време на алгоритъма.

Решения при Допълнително време (Overhead)

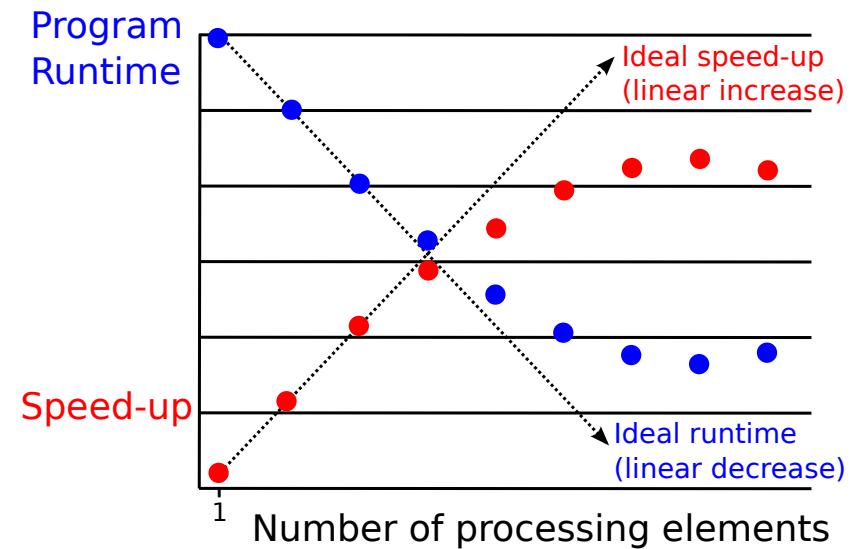
- ❖ Намаляне на броя организационни дейности;
- ❖ Увеличаване на обемите от данни обработвани от един процес;
- ❖ И др.

Забавяне (*Parallel slowdown*)

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.



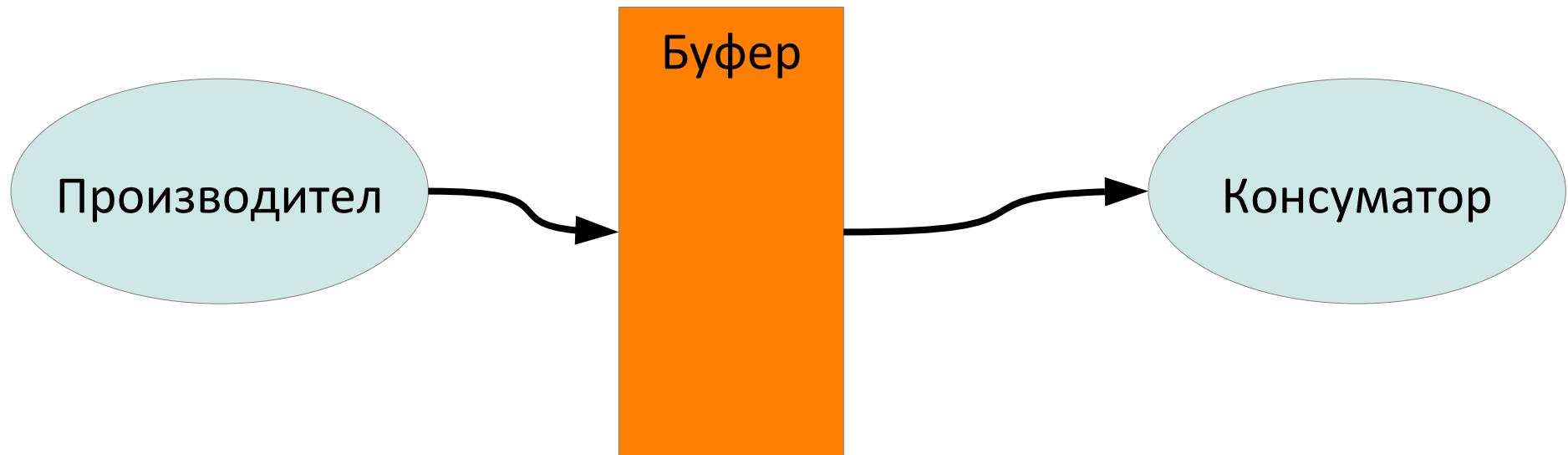
Решения при Забавяне (Parallel slowdown)

- ❖ Откриване на тесните места;
- ❖ Анализ на комуникацията между процесите;
- ❖ Отстраняване на проблема;

- ❖ Някои алгоритми като класа “неприлично паралелни” нямат този проблем, защото нямат изобщо или почти нямат комуникация;

Producer-Consumer

Производител-Консуматор



Наивно "решение" – може да доведе до мъртва хватка

```
int itemCount = 0;
procedure producer() {
    while (true) {
        item = produceItem();
        if (itemCount == BUFFER_SIZE) { sleep(); }
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if (itemCount == 1) { wakeup(consumer); }
    }
}
procedure consumer() {
    while (true) {
        if (itemCount == 0) { sleep(); }
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        if (itemCount == BUFFER_SIZE - 1) { wakeup(producer); }
        consumeItem(item);
    }
}
```



Решение със Семафори

```
semaphore fillCount = 0; // items produced
semaphore emptyCount = BUFFER_SIZE; // remaining space
procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
        putItemIntoBuffer(item); // Problem when producers are more than one
                                // Solution - use mutex m: down(m); put; up(m);
        up(fillCount);
    }
}
procedure consumer() {
    while (true) {
        down(fillCount);
        item = removeItemFromBuffer(); // Producers > 1: down(m); rem; up(m);
        up(emptyCount);
        consumeItem(item);
    }
}
```

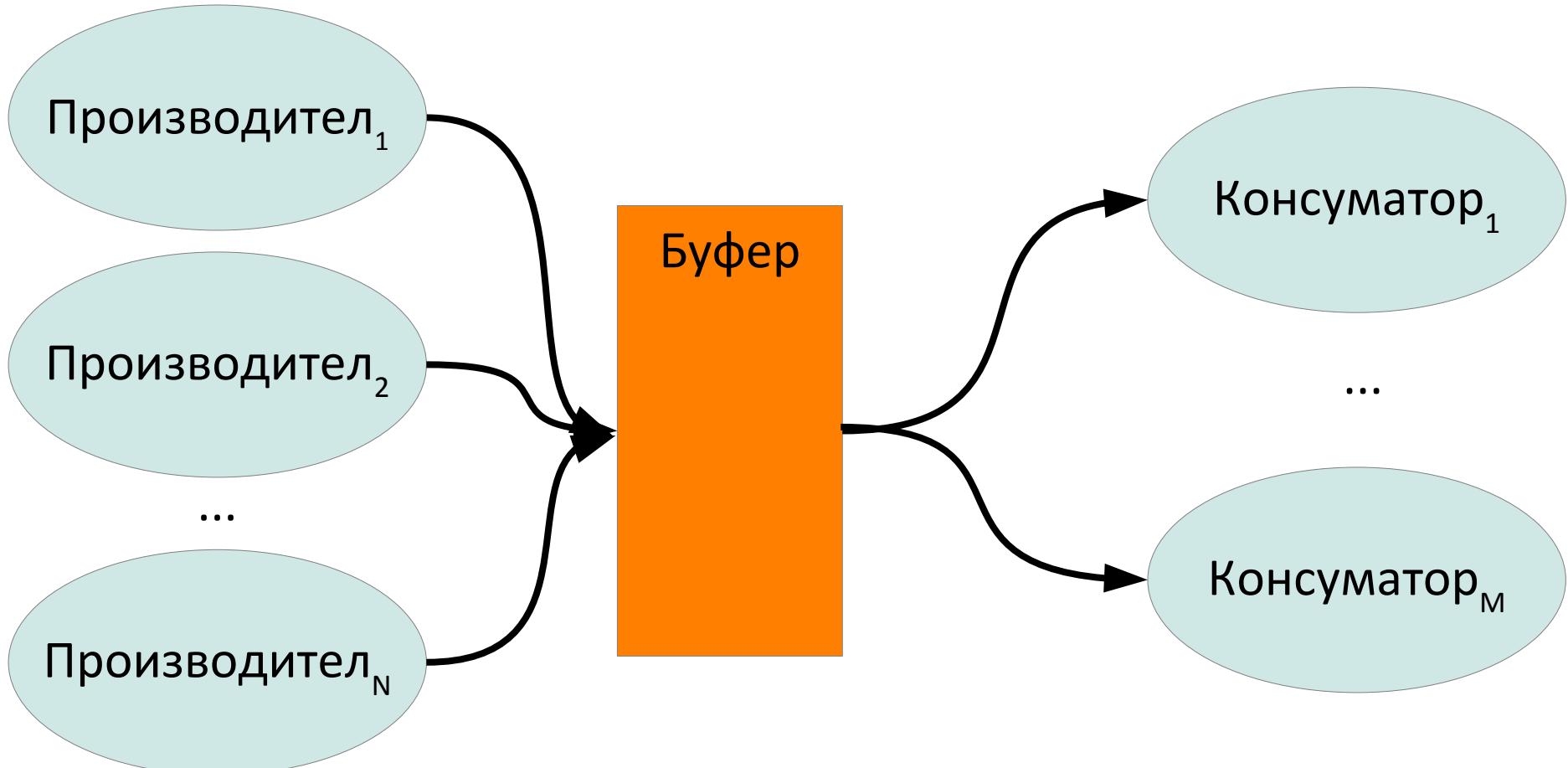


Решение без семафори

```
volatile unsigned int produceCount = 0, consumeCount = 0;
TokenType sharedBuffer[BUFFER_SIZE];
void producer(void) {
    while (1) {
        while (produceCount - consumeCount == BUFFER_SIZE) {
            schedulerYield(); /* sharedBuffer is full */
        }
        /* Write to sharedBuffer _before_ incrementing produceCount */
        sharedBuffer[produceCount % BUFFER_SIZE] = produceToken();
        /* Memory barrier required here to ensure update of the sharedBuffer is
         * visible to other threads before the update of produceCount */
        ++produceCount;
    }
}
void consumer(void) {
    while (1) {
        while (produceCount - consumeCount == 0) {
            schedulerYield(); /* sharedBuffer is empty */
        }
        consumeToken(&sharedBuffer[consumeCount % BUFFER_SIZE]);
        ++consumeCount;
    }
}
```



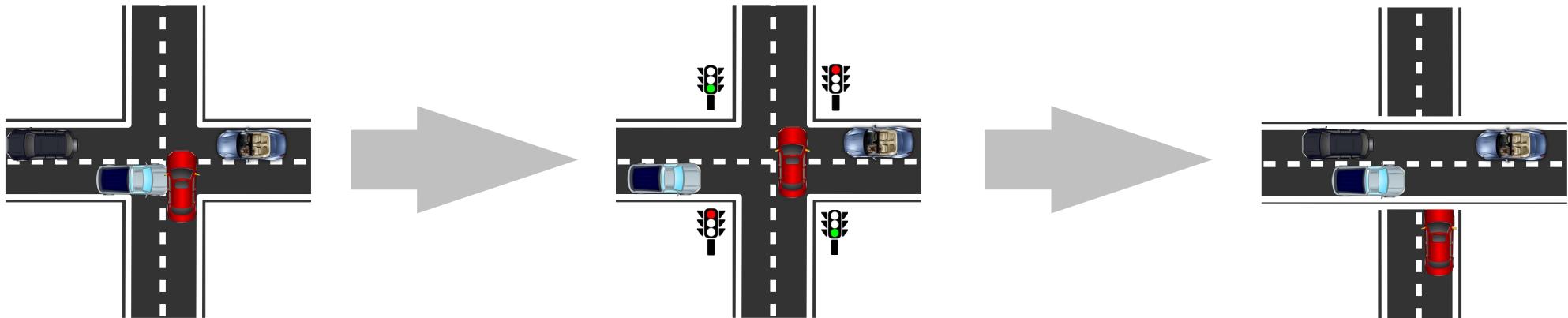
Производител-Консуматор



Други Решения на Проблемите на Паралелните Алгоритми

Други Решения

- ❖ Използване на свободни от заключване структури;
- ❖ Използване на високо паралелни алгоритми;
- ❖ Транзакционална памет (Transactional memory);
- ❖ Read-Copy-Update (RCU);
- ❖ ...;



Въпроси?

apenev@uni-plovdiv.bg



Паралелно Програмиране

Модели за паралелно програмиране.
Координация в паралелните алгоритми.

доц. д-р Александър Пенев

Модели за Паралелно Програмиране

Модели за Паралелно Програмиране

❖ Споделена памет (Shared memory)

Задачите използват общо адресно пространство (обща памет), в която могат да пишат и четат асинхронно. Използват се различни механизми за защита и контрол на достъпа до общата памет

❖ Нишки (Threads)

Нишките са подпрограми в главната програма. Те комуникират помежду си през глобална памет. Примери за този модел са Posix Threads, OpenMP и др,

❖ Предаване на съобщения (Message Passing)

Множество от задачи, използващи своя собствена локална памет за изчисленията. Обмен на данни става чрез изпращане и получаване на съобщения. Трансфера на данни обикновено е кооперативен т.е. изпращащата страна трябва да има получател. Пример: MPI

❖ Неявно взаимодействие (Implicit interaction)

В неявен модел, никакво взаимодействие между процесите не е видимо за програмиста. Вместо това компилаторът и/или изпълнителят е отговорен за неговото изпълнение

❖ Данново паралелен (Data Parallel)

Множество от задачи, работещи колективно върху обща структура от данни. Всяка задача извършва едно и също действие със своята част от данните

❖ Хибриден (Hybrid)



Модели за Паралелно Програмиране

- ❖ Една програма много данни (Single Program Multiple Data, SPMD);
- ❖ Много програми много данни (Multiple Program Multiple Data, MPMD);



Модели за паралелно програмиране

Моделите за паралелно програмиране съществуват като още една абстракция над архитектурите на хардуера и паметта.

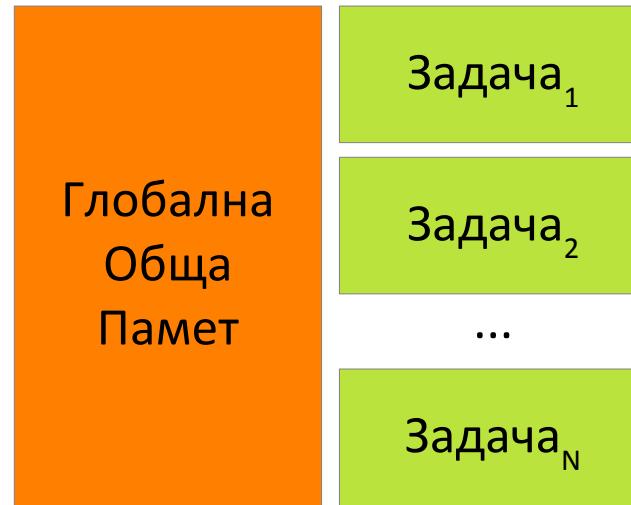
Те не са свързани с конкретен хардуер или архитектура и могат да бъдат реализирани безпроблемно върху всяка една;

Споделена памет (Shared memory)

- ❖ В този модел задачите споделят общо адресно пространство (обща памет), в което те четат и записват асинхронно;
- ❖ За контрол на достъпа до споделената памет могат да се използват различни механизми като заключване и семафори;
- ❖ Предимство на този модел от гледна точка на програмиста е, че липсва понятието „собственост“ на данни, така че не е необходимо да се уточнява изрично комуникацията на данните между задачите. Разработването на програми е максимално опростено;
- ❖ Важен недостатък (от гледна точка на производителността) е, че става трудно да се разбере и управлява местоположението на данните;

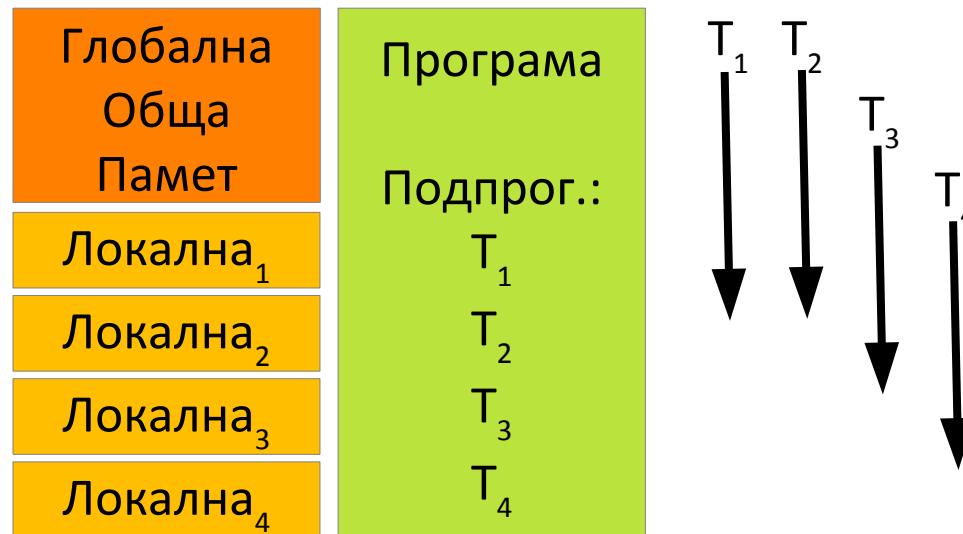
Споделена памет – реализация

- ❖ В платформите за споделена памет компилаторите превеждат променливите на програмата в действителни адреси на паметта, които са глобални;



Нишки (*Threads*)

- ❖ В много нишковия модел на паралелно програмиране, един процес може да има множество, едновременни пътища за изпълнение;
- ❖ Нишките са подпрограми в главната програма. Те комуникират помежду си през глобална памет;
- ❖ Примери за този модел са Posix Threads, OpenMP и др.;



Нишки (*Threads*)

- ❖ Основната програма се изпълнява от ОС;
- ❖ Изпълнява някаква последователна (серийна) работа, след това създава задачи (нишки), които могат да бъдат планирани и изпълнявани от ОС едновременно;
- ❖ Всяка нишка има локални данни, но също така споделя всички ресурси програмата;
- ❖ Работата на нишката може най-добре да се опише като подпрограма в основната програма. Всяка нишка може да изпълни всяка подпрограма едновременно с другите нишки;
- ❖ Нишките комуникират помежду си чрез глобална памет, което изисква конструкции за синхронизация, за да се гарантира, че повече от една нишка не актуализира един и същ глобален адрес в даден момент;
- ❖ Нишките могат да се създават и унищожават по време на прог.;

Нишки (Threads) – реализация

От гледна точка на програмирането, реализациите на нишки обикновено включват:

- ❖ Библиотека от подпрограми (методи), които се извикват от паралелен изходен код;
- ❖ Набор директиви за компилатора, вложени в сериен или паралелен изходен код;

И в двета случая програмистът е отговорен за определянето на целия паралелизъм.

Предаване на съобщения (Message Passing)

- ❖ Множество от задачи, използващи своя собствена локална памет за изчисленията;
- ❖ Обмен на данни става чрез изпращане и получаване на съобщения;
- ❖ Трансфера на данни обикновено е кооперативен т.е. изпращащата страна трябва да има получател;
- ❖ Пример: MPI



Предаване на съобщения – реализация

- ❖ Реализира се като библиотека от функции, които се използват в изходния код на паралелните програми;
- ❖ Програмистът отговаря за определянето на целия паралелизъм;
- ❖ През 80-те години на миналия век възникват множество библиотеки за предаване на съобщения. Това са различни (и несъвместими помежду си) реализации;
- ❖ През 1992 г. е създаден **MPI forum** с основната цел да се създаде стандартен интерфейс за предаване на съобщения;
- ❖ MPI (и MPI-2) е "фактически" индустриален стандарт за предаване на съобщения в паралелни приложение (на НРС);
- ❖ За архитектури със споделена памет, MPI реализациите обикновено не използват мрежова комуникация, а вместо това те използват споделена памет за по-добра производителност;

Неявно взаимодействие (Implicit interaction)

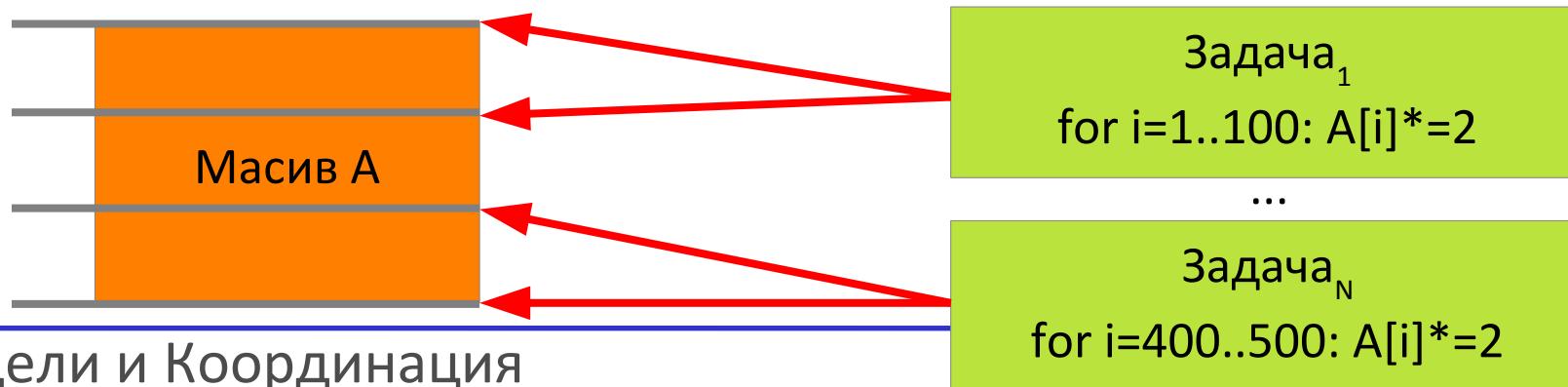
- ❖ В неявен модел, никакво взаимодействие между процесите не е видимо за програмиста;
- ❖ Вместо това компилаторът и/или изпълнителят е отговорен за неговото изпълнение;
- ❖ Примери за този вид модел може да се видят в някои домейн-специфични езици и функционалните езици;
- ❖ Този тип паралелизъм се контролира трудно;

Неявно взаимодействие (Implicit interaction)

- ❖ Други примери за този модел са:
 - ❖ Автоматичната паралелизация на ниво компилатор;
 - ❖ Супер-скаларното изпълнение;
 - ❖ ILP;

Данново паралелен (Data Parallel)

- ❖ По-голямата част от паралелната работа се фокусира върху извършването на операции върху набор от данни;
- ❖ Наборът от данни обикновено е организиран в обща структура, като например масив;
- ❖ Набор от задачи работят съвместно върху една и съща структура на данни, но всяка задача работи на различен дял от тази структура;
- ❖ Задачите изпълняват една и същата операция над своя дял от работата, например "умножете с 2 всеки елемент от масив".



Данново паралелен (Data Parallel)

- ❖ В архитектурите със споделена памет всички задачи имат достъп до структурата от данни, чрез глобална памет;
- ❖ В разпределените архитектури на паметта структурата от данни е разделена и се намира като "парчета" в локалната памет на всяка задача;

Даново паралелен – реализация

- ❖ Програмирането с даново паралелен модел, обикновено се осъществява чрез писане на програма с даново паралелни конструкции;
- ❖ Те са или част от езика и съответно компилатора ги превежда (много често до MPI извиквания), или са реализирани както библиотеки съдържащи класове и методи/функции от даново паралелни типове/конструкции;

*Хибриден модел (*Hibrid*)*

- ❖ Комбинация от два или повече от другите модели;
- ❖ Например: MPI+OpenMP или MPI+Pthreads или MPI+DataParallel;

Координация в Паралелните Алгоритми

Видове координация / синхронизация (на процеси и на данни)

- ❖ **Бариери (Barrier)**

Бариера в сурс кода за група нишки или процеси се мястото, в което всички нишки/процеси трябва да спрат, докато всички други (от групата) не достигнат тази “бариара”;

- ❖ **Заключвания/Семафори (Lock/Semaphore)**

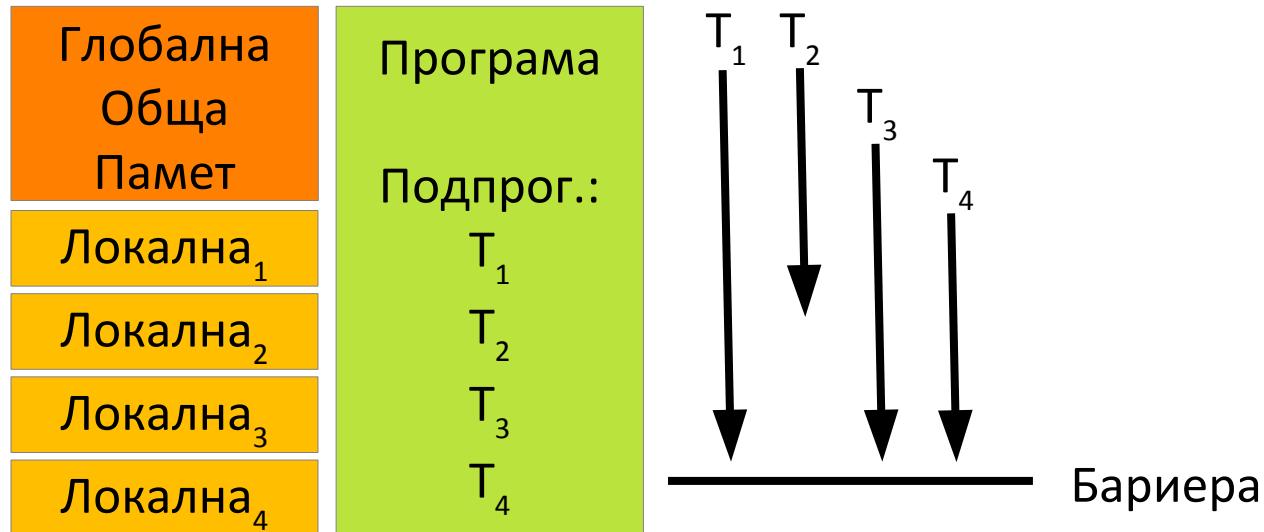
Обикновено се използва за сериализиране (защита) на достъпа до глобални данни или общ код;

- ❖ **Синхронни комуникационни операции
(Synchronous communication operations);**

Бариери (Barrier)

- ❖ Обикновено се предполага, че всички задачи се изпълняват;
- ❖ Всяка задача изпълнява своята работа, докато достигне бариерата;
- ❖ След това задачата спира или "блокира";
- ❖ Когато последната задача достигне бариерата, всички задачи са синхронизирани;
- ❖ Какво става след бариерата може да варира в зависимост от реализацията, например може да се продължи с изпълнението на последователната (серийната) част от кода; може задачите да продължат работата си и др.;

Бариери (Barrier)



Барьеры (Barrier) – пример (C++11)

```
void DoWork() {
    Tasks& tasks;
    int n_threads;
    vector<thread*> workers;

    barrier task_barrier(n_threads);

    for (int i = 0; i < n_threads; ++i) {
        workers.push_back(new thread([&] {
            bool active = true;
            while(active) {
                Task task = tasks.get();
                // perform task
            }
        }));
    }

    task_barrier.wait();
}
```

Барьеры (Barrier) – пример (C++11)

```
// perform task
...
task_barrier.arrive_and_wait();
}
});
}

// Read each stage of the task until all are complete.
while (!finished()) {
    GetNextStage(tasks);
}
}
```

Заключвания/Семафори (*Lock/Semaphore*)

- ❖ Може да участват произволен брой задачи;
- ❖ Обикновено се използва за сериализиране (защита) на достъпа до глобални данни или общ код;
- ❖ Само една задача в даден момент може да използва (притежава) заключването / семафора / флага;
- ❖ Първата задача, която “заключи” променливата и задава стойност, след това тази задача може безопасно (серийно) да получи достъп до защитените данни или код;
- ❖ Други задачи могат да се опитат да придобият заключване, но трябва да изчакат, докато задачата, която “държи” заключващо го освободи;

Заключване – пример (C#)

```
using System; using System.Threading;
class Program {
    static readonly object _object = new object();
    static void A() {
        lock (_object) {
            Thread.Sleep(100);
            Console.WriteLine(Environment.TickCount);
        }
    }
    static void Main() {
        for (int i = 0; i < 10; i++) {
            ThreadStart start = new ThreadStart(A);
            new Thread(start).Start();
        }
    }
}
```



Заключвания/Семафори (*Lock/Semaphore*)

- ❖ Има блокиращи и не блокиращи семафори;
- ❖ Има “бинарни” семафори, наречени мутекс (mutex);
- ❖ В общия случай семафора съдържа броя на свободните ресурси, ако те са повече от един, като при използване (acquire) променливата се намаля с 1, а при освобождаване (release) се увеличава с 1. Блокирането на задачи е когато ресурси вече няма т.е. семафора е 0;
- ❖ Реализацията на семафори и други подобни, обикновено използва специални (атомарни) инструкции (от тип Test-and-Set) в съвременните процесори, като например BTS, CMPXCHG и др.;

Синхронни комуникационни операции

- ❖ Включва само онези задачи, изпълняващи комуникационна операция;
- ❖ Когато дадена задача извършва комуникационна операция е необходима никаква форма на координация с другата задача(и), участващи в комуникацията. Например, преди дадена задача може да извърши операция за изпращане, първо трябва да получи потвърждение от получаващата задача, че е добре да изпрати;

Въпроси?

apenev@uni-plovdiv.bg



Паралелно Програмиране

Дизайн на паралелни програми

доц. д-р Александър Пенев

Първи Стъпки

Първи стъпки

❖ Разбиране на проблема

Подробно запознаване с проблема, който ще се решава „паралелно“. Ако имаме вече написана „серийна“ (не паралелна) програма, която ще преработваме тяба добра да разберем и кода;

❖ Преди да инвестираме време в преработка

Добре е да направим груба оценка на това, какво може да се постигне в най-добрия случай т.е. дали проблема може да е паралелен като ray tracer или е по-скоро като „Фибоначи“?

❖ Определяне на т.нар. „горещи точки“ (hotspots) на програмата

Това са местата на програмата където се извършва най-много „работка“. В повечето случаи това са няколко места в програмата. Може да се използват profiler-и и анализ на производителността за да ги открием, след което да се фокусираме в разпаралеляването точно на тях;

❖ Определяне на т.нар. „тесни места“ (bottlenecks) в програмата

Има ли области, които са непропорционално бавни или предизвикват спиране или отлагане на паралелна работа? Например В/И обикновено е нещото, което забавя програмата. Възможно е да се преструктурира програмата или да се използва различен алгоритъм за намаляване или премахване на ненужните бавни области;

❖ Идентифицирайте пречките пред паралелизма

Един общ клас пречки са данновите зависимости. Това беше показано от примера с пресмятането на числата на Фибоначи;



Първи стъпки

- ❖ **Проучете други алгоритми, ако е възможно**

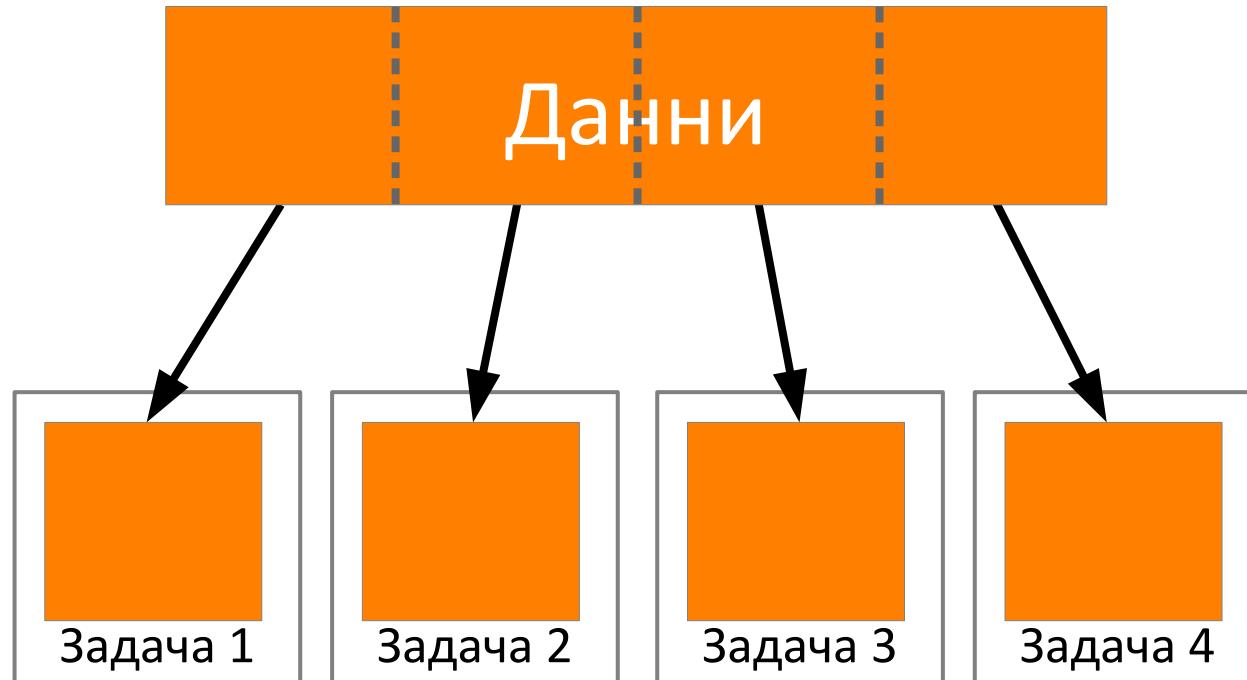
Това може да е най-голямо значение при проектирането на паралелно приложение;

- ❖ **Възползвайте се от оптимизиран софтуер за трети страни**

Съществуват много добре оптимизирани и силно паралелни библиотеки – математически библиотеки, налични от водещи доставчици (IBM ESSL, MKL на Intel, AMCL на AMD и др.); Фреймурци, библиотеки с паралелни структури от данни и алгоритми и др.;

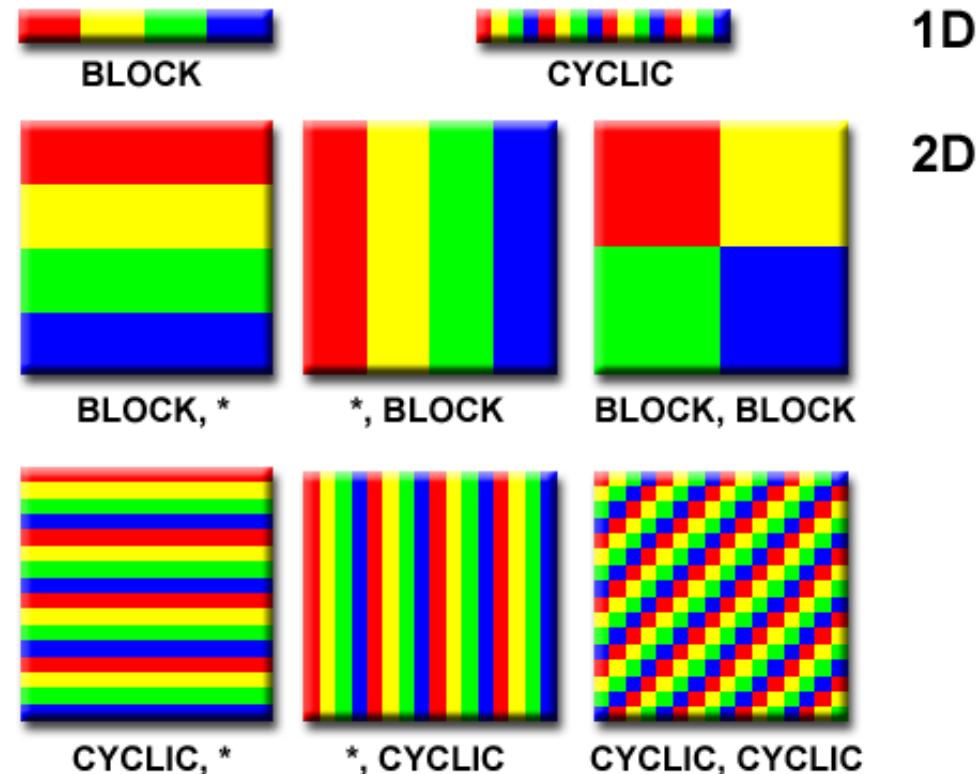
Как ще се Декомпозират Данните

Разбиване на данните при даннов паралелизъм



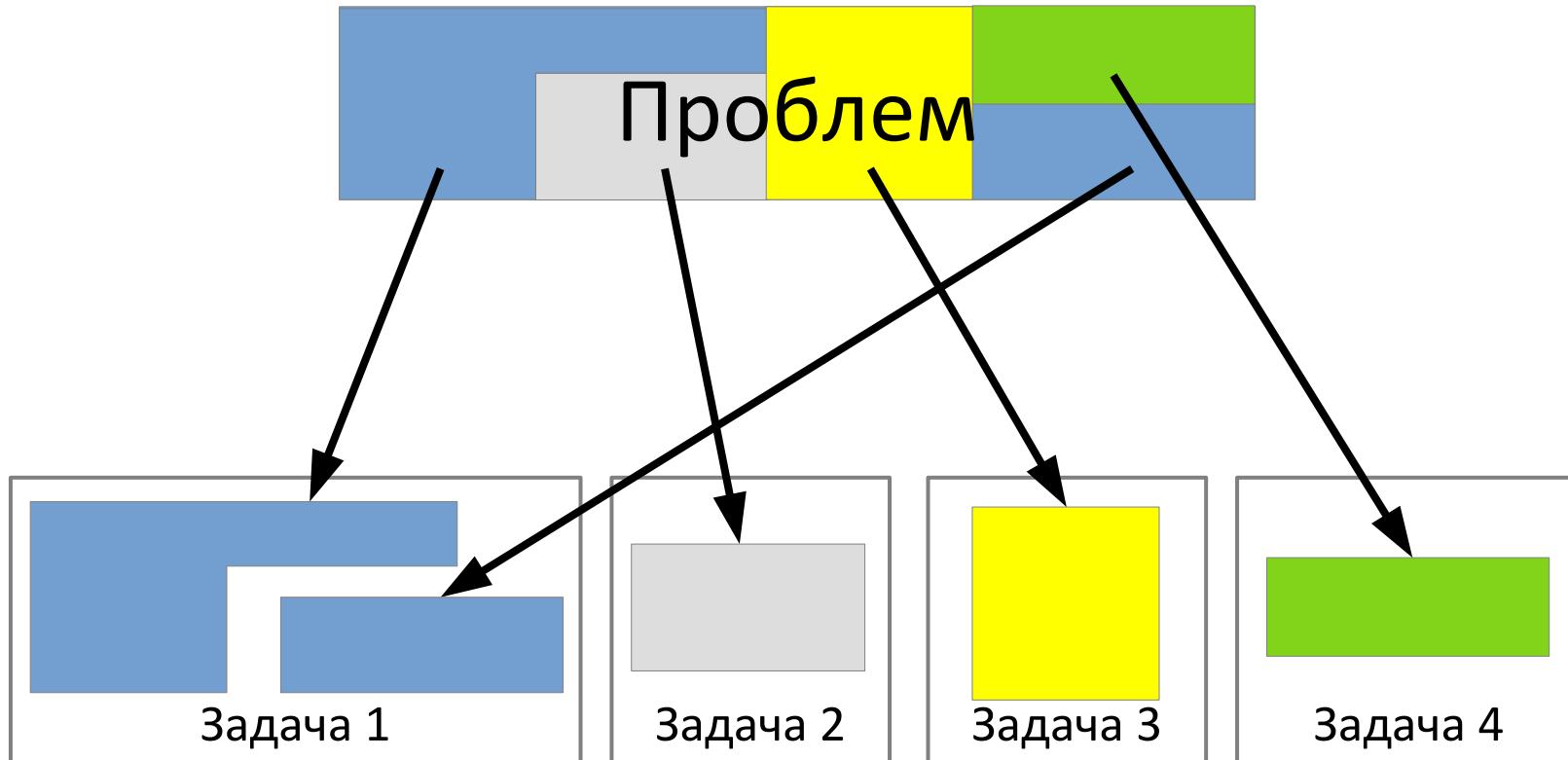
Видове декомпозиция на данни

- ❖ 1D, 2D, 3D, ...;
- ❖ Равномерно и Неравномерно;
- ❖ Статично и Динамично;
- ❖ Блоково или Циклично;



Как ще се Декомпозира Алгоритъма

Декомпозиране на алгоритъма



Видове Комуникация Между Подзадачите

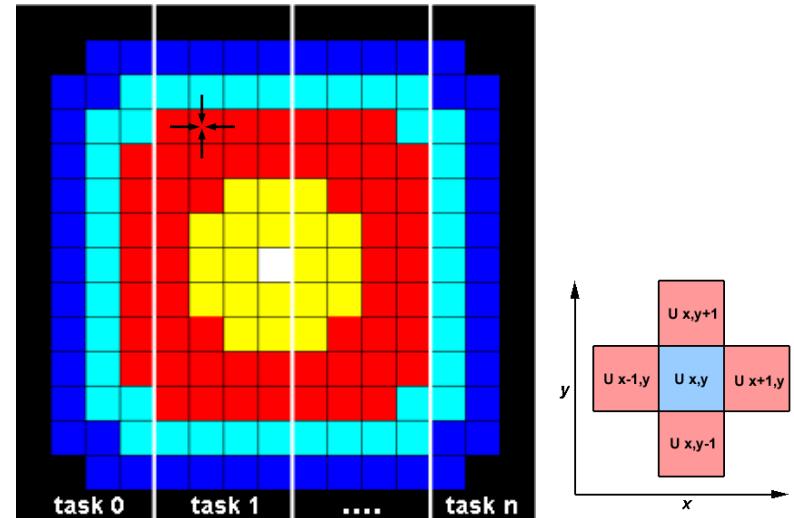
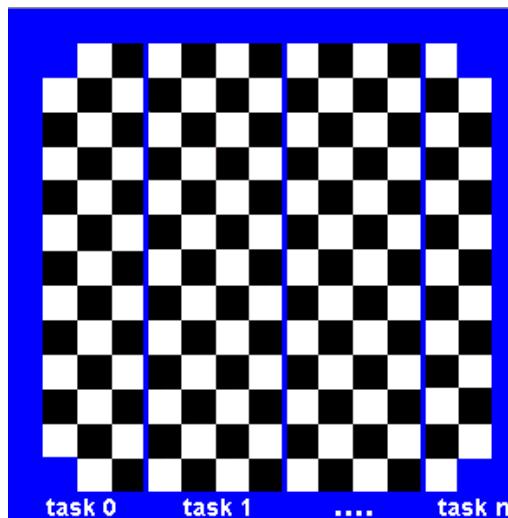
Комуникация между подзадачите

- ❖ Без комуникация – например да обърнем цвета от черен в бял и от бял в черен за всяка точка

Това не е свързано с дннова зависимост между различните части на изображението и това позволява да разбием данните на и да обработим парчетата независимо едно от друго;

- ❖ С комуникация – например да направим Gaussian Blur на дадено изображение

При разбиване на изображението, може да обработим всяко парче паралелно с другите, но при някои точки (граничните) има зависимост от данните на съседните парчета;



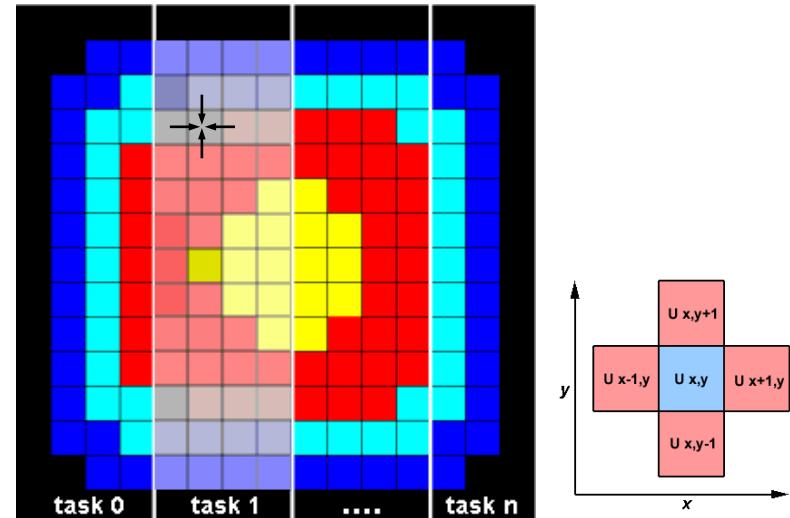
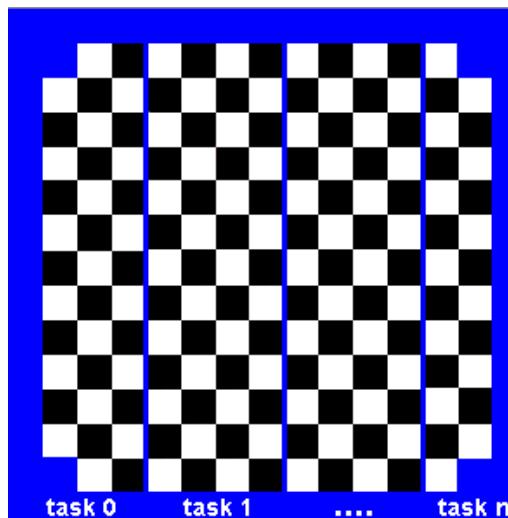
Комуникация между подзадачите

- ❖ Без комуникация – например да обърнем цвета от черен в бял и от бял в черен за всяка точка

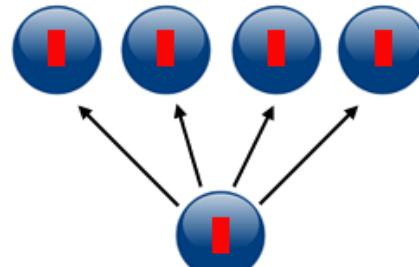
Това не е свързано с дннова зависимост между различните части на изображението и това позволява да разбием данните на и да обработим парчетата независимо едно от друго;

- ❖ С комуникация – например да направим Gaussian Blur на дадено изображение

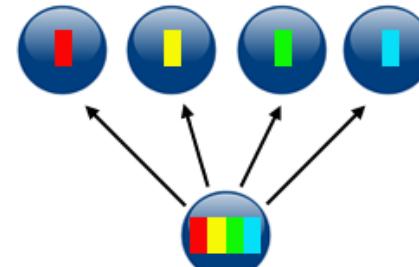
При разбиване на изображението, може да обработим всяко парче паралелно с другите, но при някои точки (граничните) има зависимост от данните на съседните парчета;



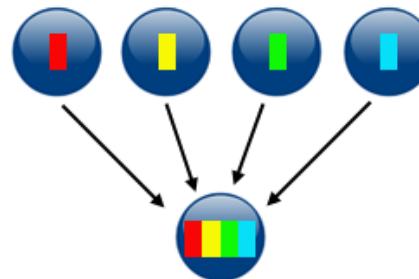
Комуникация между подзадачите



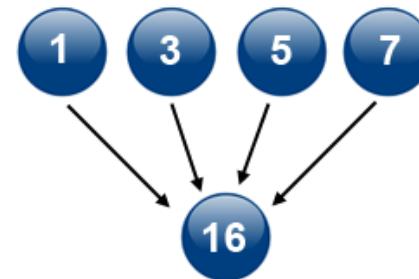
Разпръскване
(Broadcast)



Разпределяне
(Scatter)



Събиране
(Gather)



Редукция
(Reduction)

Комуникация между подзадачите

❖ Разпръскване (Broadcast)

При този тип комуникация между задачите (обикновено една) задача предава едни и същи данни/команди към всички подзадачи, които те в последствие трябва да обработят/изпълнят;

❖ Разпределение (Scatter)

При този тип комуникация между задачите (обикновено една) задача разпределя данните/командите на части и предава всяка част на различна подзадача;

❖ Събиране (Gather)

Обработените данни се връщат от всички подзадачи на главната (обикновено на тази, която е разпределила преди това работата) и от тях се формира общия резултат. Това обикновено става без съществени допълнителни обработки;

❖ Редукция (Reduction)

Резултатите от обработените данни се връщат от всички подзадачи на главната (обикновено на тази, която е разпределила преди това работата) и от тях се формира общия резултат чрез т. нар. Редукция – сравнително прост или по-сложен алгоритъм (най-често) обобщаващ крайния резултат на базата на под резултатите. Например, ако търсим максимум в голям масив от данни, те може да се разпределят на части и всяка подзадача да търси максимума в своята част от данните, след това максимумите се редуцират в главната задача, като максимум от тези максимуми. Много често се формират дървета на редукция т.е. редукция на повече нива;

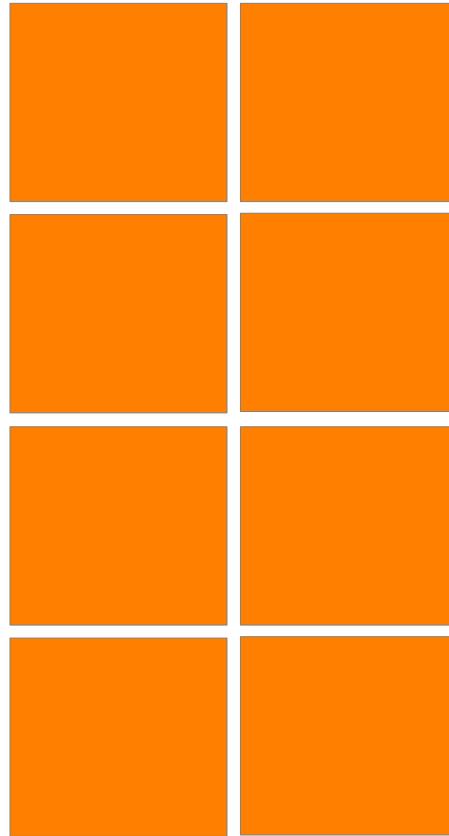
❖ Други



Вход / Изход

Вход / Изход

Процесори (P)



В/И Пътища

Дискове (D)

Вход / Изход

❖ Проблеми:

- ❖ В/И операциите обикновено са пречещи на паралелизма;
- ❖ Операциите за В/И изискват по-голямо количество време от операциите с оперативната памет;
- ❖ Паралелните В/И системи може да не са добре реализирани или или да не са налични за всички платформи;
- ❖ Операциите по запис могат да доведат до презаписване на файловете, когато всички задачи имат достъп до едно и също файлово пространство;
- ❖ Операциите за четене могат да бъдат повлияни от способността на файловия сървър да обработва няколко заявки за четене едновременно;
- ❖ В/И, които минава през мрежата (NFS, не-локални), могат да причинят сериозни проблеми;



Вход / Изход

- ❖ Налични са паралелни файлови системи. Например:
 - ❖ GPFS: Паралелна файлова система (IBM). Сега се нарича IBM Spectrum Scale;
 - ❖ Luster: за Linux кълстери (Intel);
 - ❖ HDFS: Разпределена файлова система Hadoop (Apache);
 - ❖ PanFS: Файлови системи Panasas ActiveScale за Linux кълстери (Panasas, Inc.);
 - ❖ и др.
- ❖ Спецификацията на паралелния В/И интерфейс за MPI е налична като част от MPI-2;

Вход / Изход

- ❖ Препоръки:
 - ❖ Намалете общия В/И максимално;
 - ❖ Ако имате достъп до паралелна файлова система, използвайте я;
 - ❖ Писането на големи парчета данни е по-ефективно от писането на малки;
 - ❖ Няколко по-големи файлове е по-добре от много по-малки файлове;
 - ❖ Ограничете В/И в една или няколко серийни части на програмата. След това използвайте паралелни комуникации за разпределение на данните към паралелните подзадачи. Това е валидно и за записа;
 - ❖ Ограничете В/И в една или няколко подзадачи;

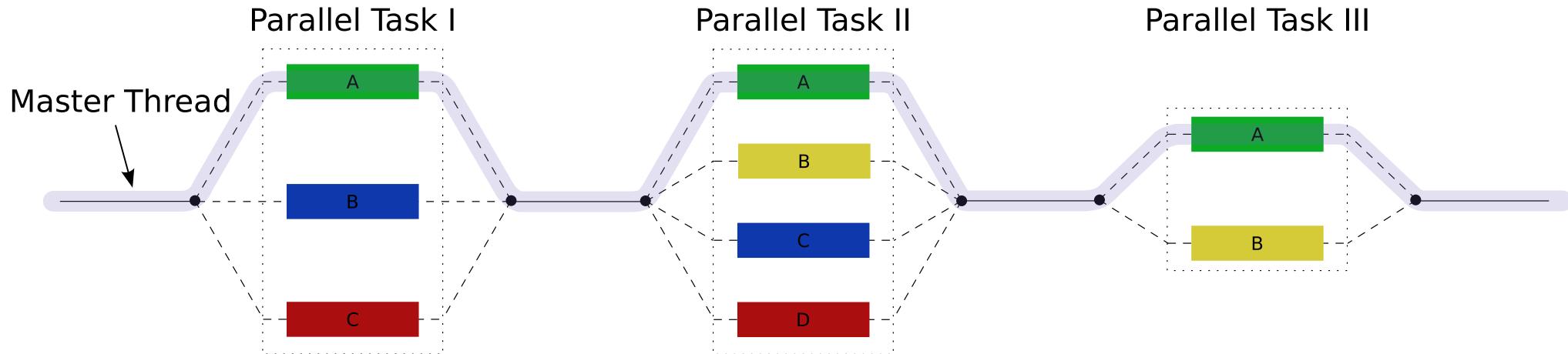
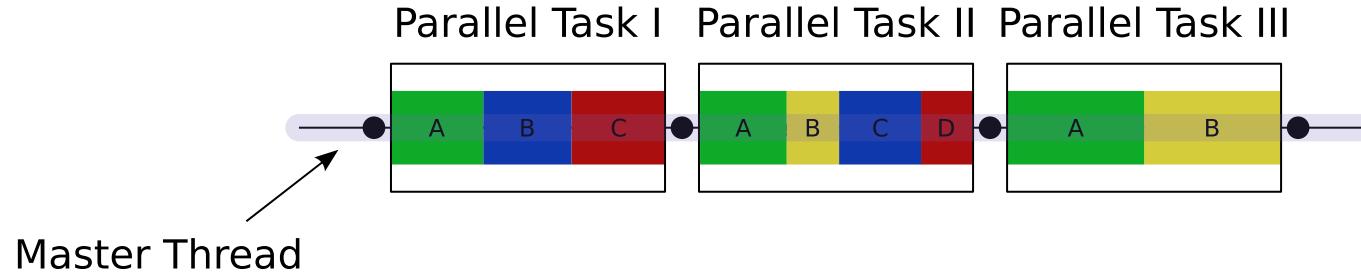
Добри Практики

Fork-Join

Fork-Join – пример

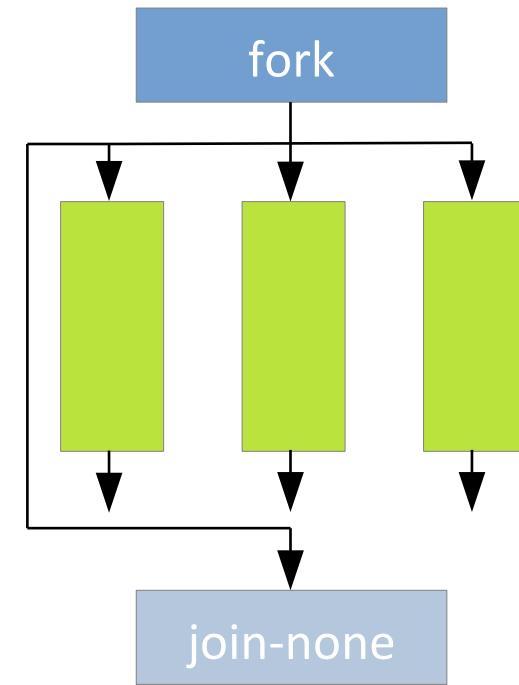
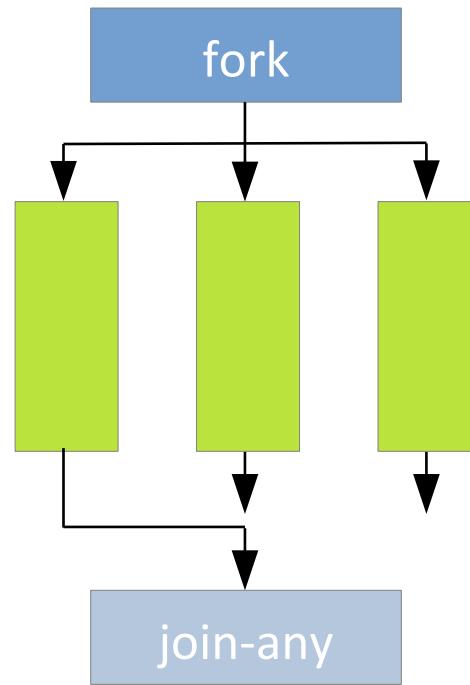
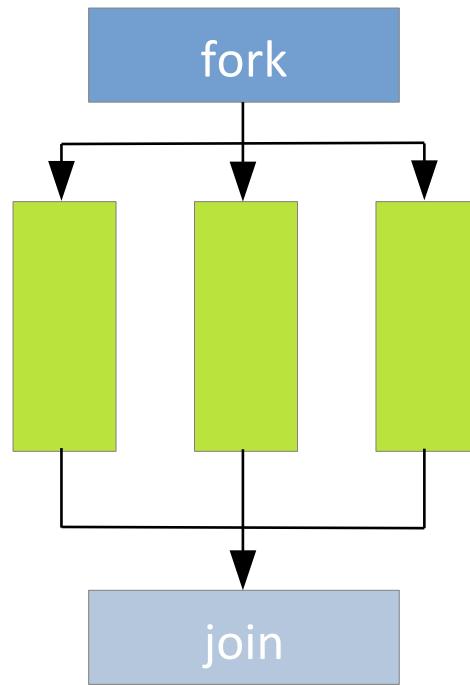
```
mergesort(A, lo, hi):
    if lo < hi:
        mid = [lo + (hi - lo) / 2]
        fork mergesort(A, lo, mid) // process in parallel
        mergesort(A, mid, hi)    // main task handles second
        join
    merge(A, lo, mid, hi)
```

Fork-Join



Fork-Join

- ❖ fork-join;
- ❖ fork-join-any;
- ❖ fork-join-none;

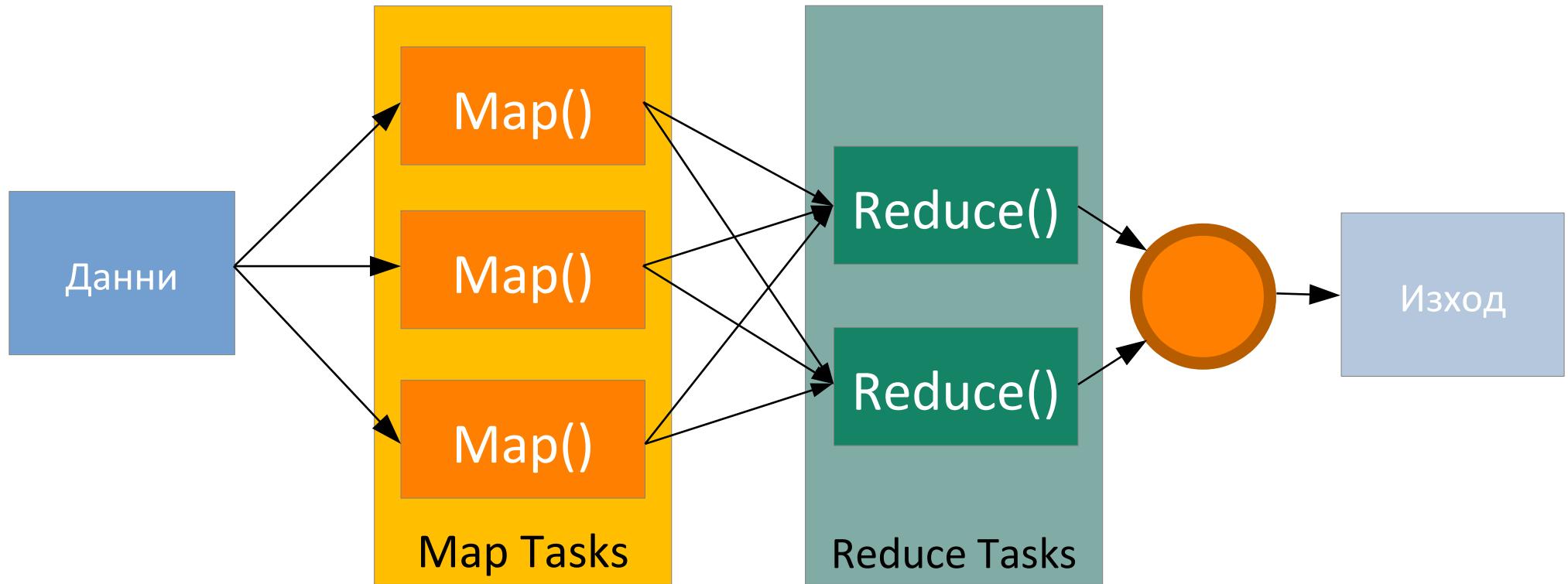


Fork-Join

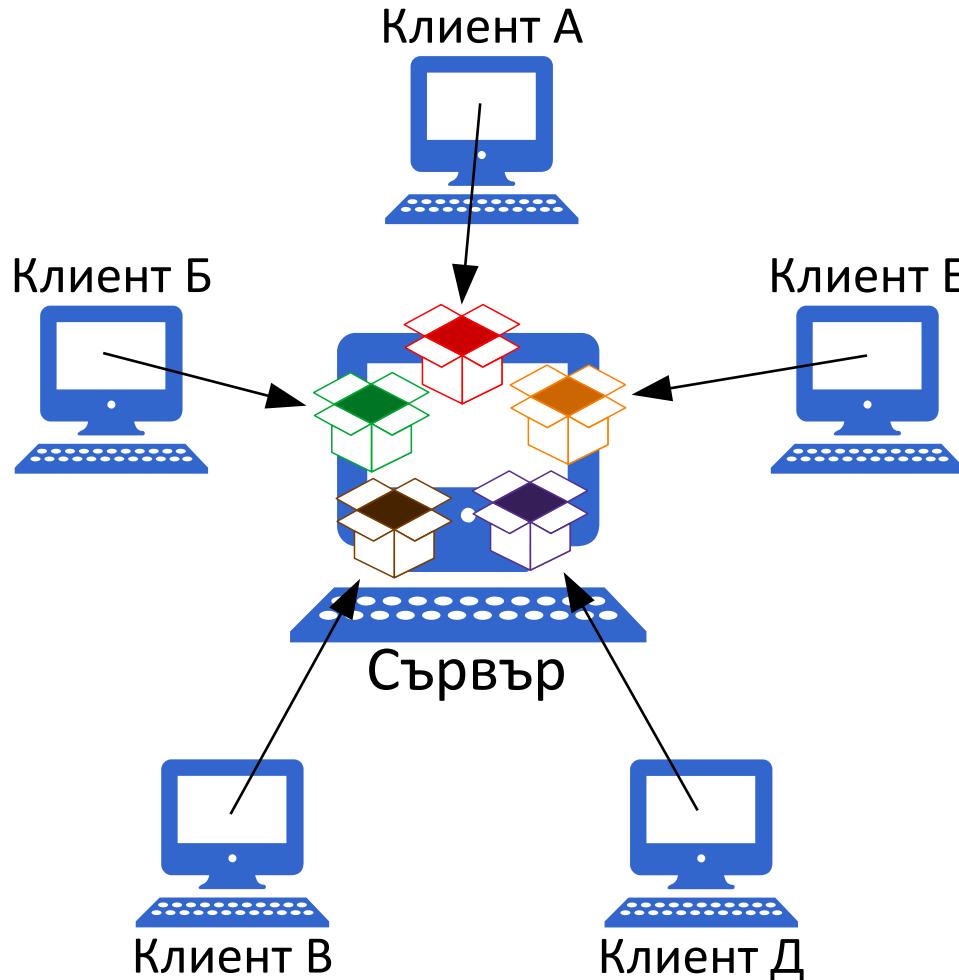
- ❖ OpenMP;
- ❖ Java concurrency framework;
- ❖ Task Parallel Library for .NET;
- ❖ Intel's Threading Building Blocks (TBB);
- ❖ Cilk;
- ❖ И др.

MapReduce

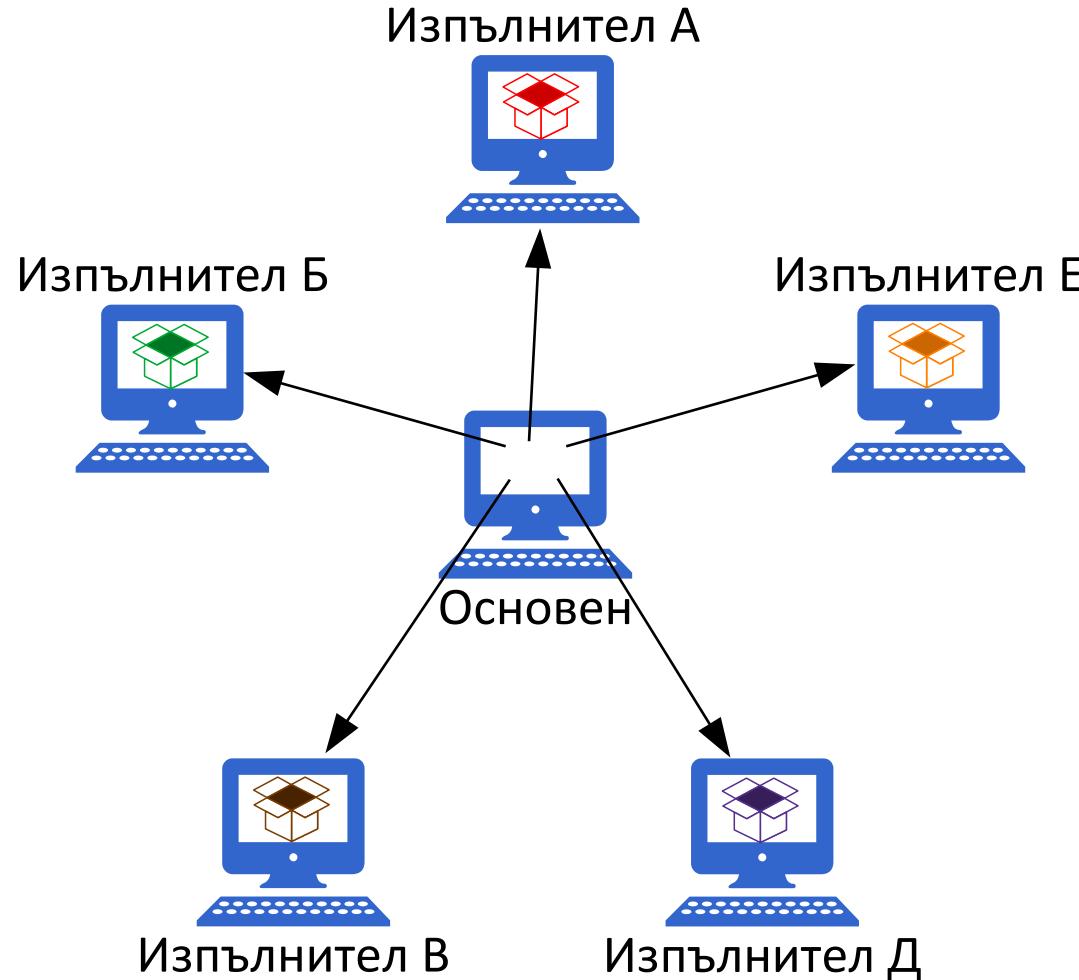
MapReduce



Класическа схема на обработка на данни



MapReduce подход



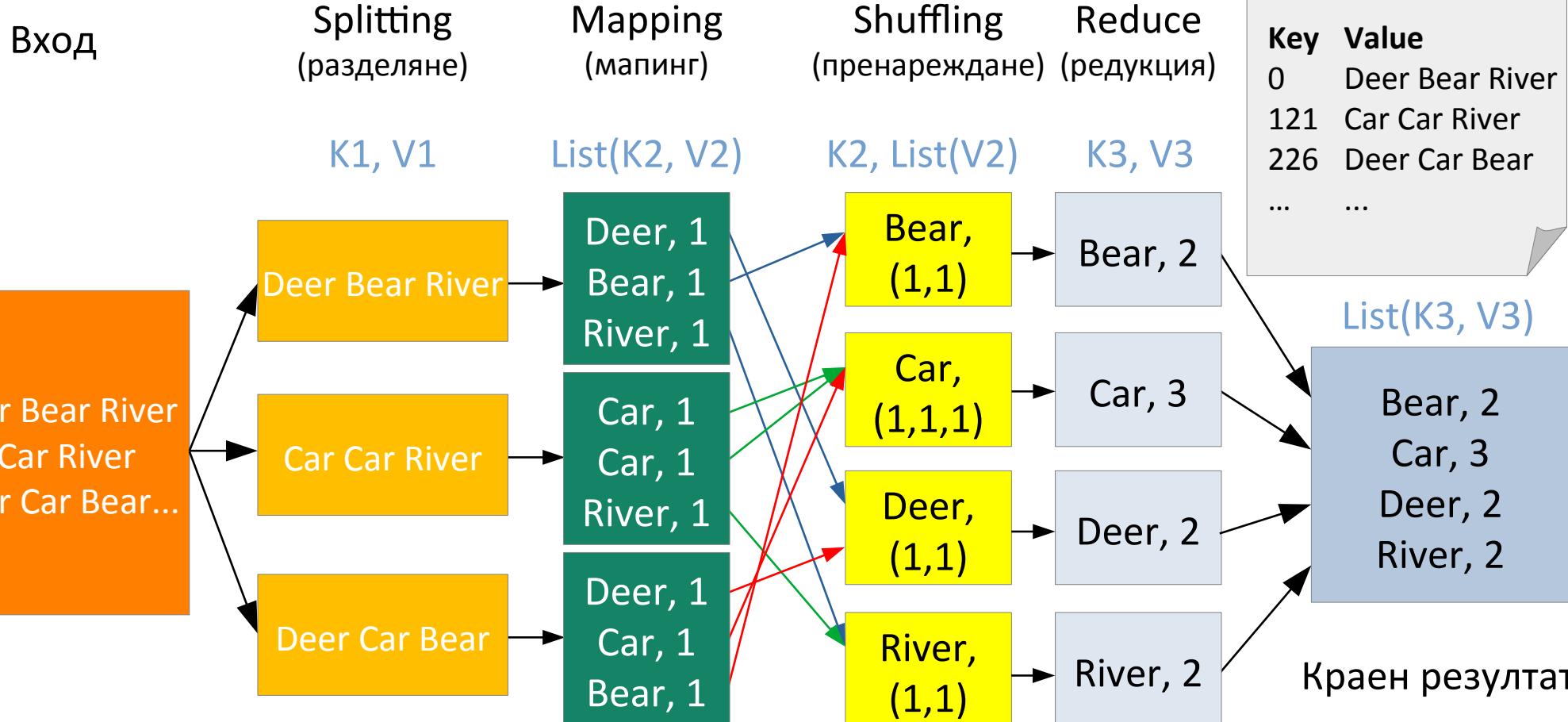
Пример – броене на думи (традиционн подход)

Входен файл

```
Deer Bear River
Car Car River
Deer Car Bear
...
```



MapReduce броене



MapReduce – пример

```
public static class Map extends  
        Mapper<LongWritable, Text, Text, IntWritable> {  
    public void map(LongWritable key, Text value,  
                    Context context)  
        throws IOException, InterruptedException {  
        String line = value.toString();  
        StringTokenizer tokenizer = new StringTokenizer(line);  
        while (tokenizer.hasMoreTokens()) {  
            value.set(tokenizer.nextToken());  
            context.write(value, new IntWritable(1));  
        }  
    }  
}
```

MapReduce – пример

```
public static class Reduce extends  
    Reducer<Text, IntWritable, Text, IntWritable> {  
    public void reduce(Text key, Iterable<IntWritable> values,  
                      Context context)  
        throws IOException, InterruptedException {  
        int sum=0;  
        for(IntWritable x: values) {  
            sum += x.get();  
        }  
        context.write(key, new IntWritable(sum));  
    }  
}
```

MapReduce – пример

```
public static void main(String[] args) throws Exception {  
    Configuration conf = new Configuration();  
    Job job = new Job(conf, "My Word Count Program");  
    job.setJarByClass(WordCount.class);  
    job.setMapperClass(Map.class);  
    job.setReducerClass(Reduce.class);  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
    job.setInputFormatClass(TextInputFormat.class);  
    job.setOutputFormatClass(TextOutputFormat.class);  
    Path outputPath = new Path(args[1]);  
    // Configuring the I/O path from the job's filesystem  
    FileInputFormat.addInputPath(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
}
```



Hotspots и Bottlenecks

Hotspots

- ❖ Това са местата в програмата, които отнемат най-много ресурси (най-често това е ресурса време);
- ❖ Инструментите за анализ на производителността на софтуер помагат за лесното им откриване;
- ❖ В тях трябва да се насочат основите усилия за оптимизация на програмата, както и при възможност да векторизираме и/или разпаралим тези части на алгоритъма;
- ❖ На следващите слайдове е показан част от анализа на приста програма за умножение на матрици. Използван е софтуерът за анализ Intel VTune Amplifier;

Hotspots

/home/student1/intel/amplxe/projects/matrix - Intel VTune Amplifier XE 2011

File View Help

Am r003lh

Lightweight Hotspots - Hotspots ⚡ ?

Intel VTune Amplifier XE 2011

Analysis Target Analysis Type Collection Log Summary Bottom-up Module Timeline

Elapsed Time: ② 108.382s

CPU Time: ② 107.877s
Instructions Retired: 8,600,000,000
CPI Rate: ② 33.442
The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify...
Paused Time: ② 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

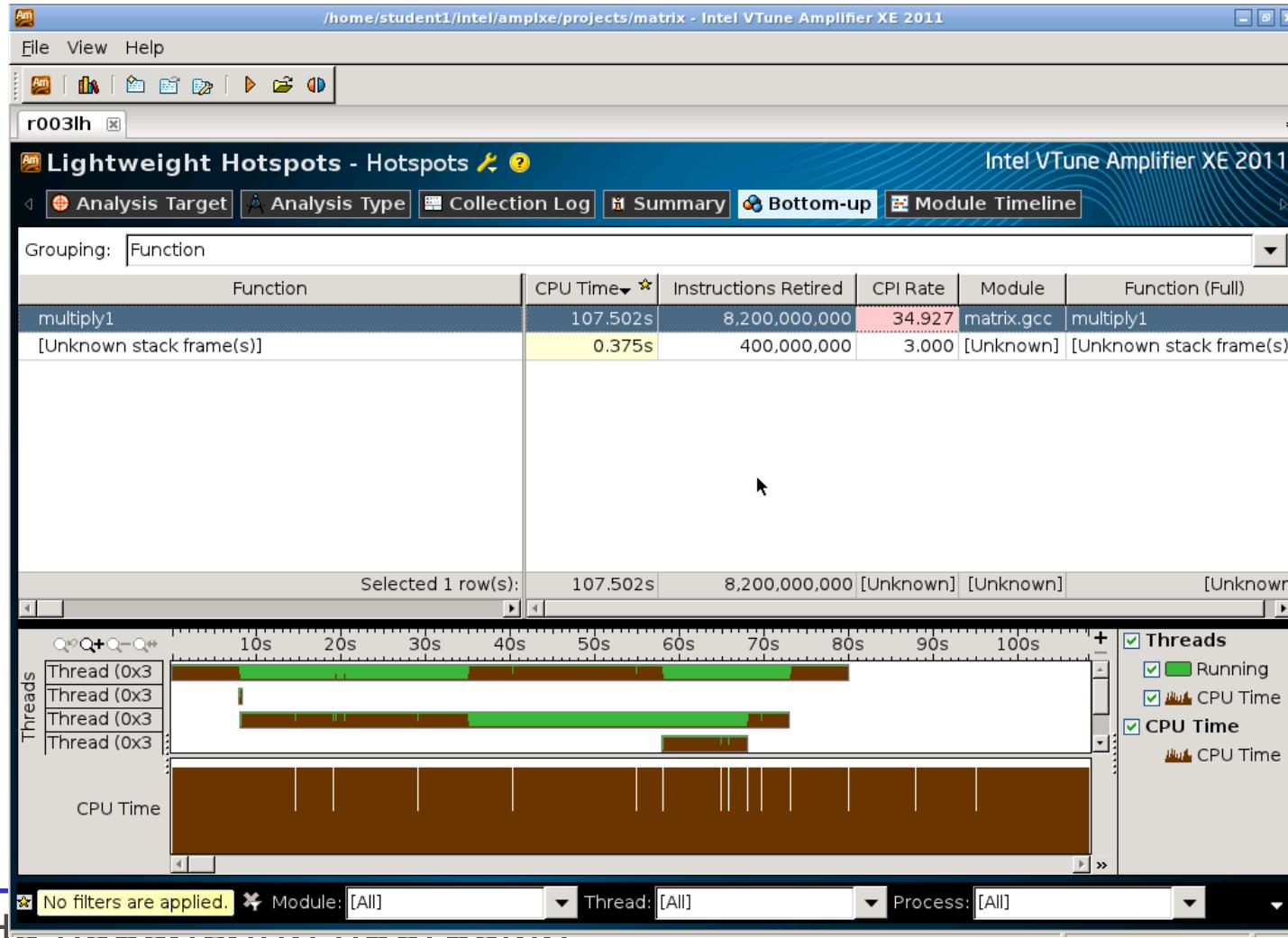
Function	CPU Time ②
multiply1	107.502s
[Unknown stack frame(s)]	0.375s

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Command Line:	/opt/intel/vtune_amplifier_xe/samples/en/C++/matrix/linux/matrix.gcc
Frequency:	2.666 GHz
Logical CPU Count:	8
User Name:	student1
Operating System:	Linux
Computer Name:	performance-tunning
Result Size:	1 MB

Hotspots



Hotspots

/home/student1/intel/amplxe/projects/matrix - Intel VTune Amplifier XE 2011

File View Help

r003lh

Lightweight Hotspots - Hotspots ?

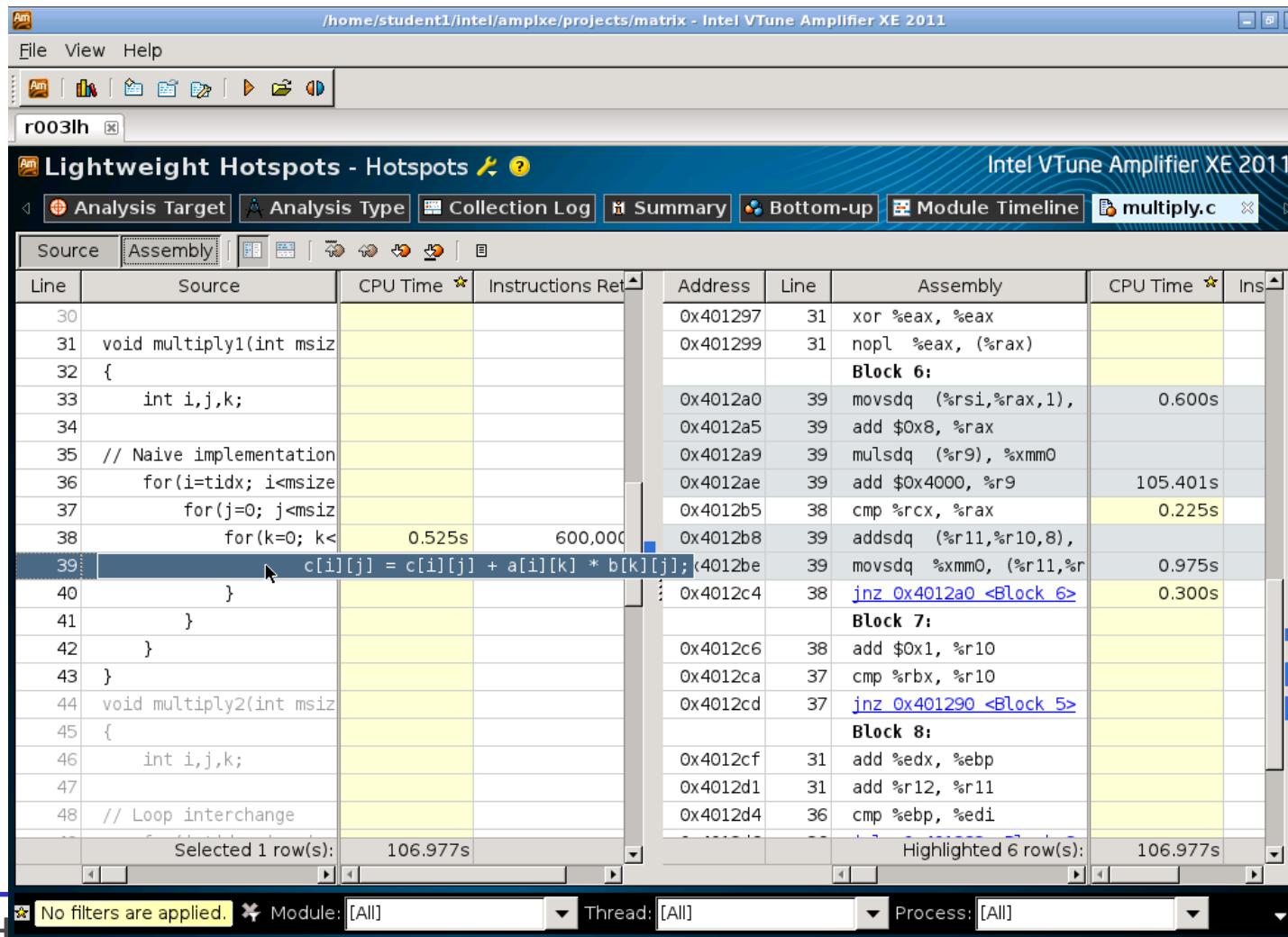
Analysis Target Analysis Type Collection Log Summary Bottom-up Module Timeline multiply.c

Line	Source	CPU Time *	Instructions Retired
30			
31	void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE		
32	{		
33	int i,j,k;		
34			
35	// Naive implementation		
36	for(i=tidx; i<msize; i=i+numt) {		
37	for(j=0; j<msize; j++) {		
38	for(k=0; k<msize; k++) {		
39	c[i][j] = c[i][j] + a[i][k] * b[k][j];	0.525s	600,000,000
40	}	106.977s	7,600,000,000
41	}		
42	}		
43	}		
44	void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE		
45	{		
46	int i,j,k;		
47			
48	// Loop interchange		

No filters are applied. Module: [All] Thread: [All] Process: [All]

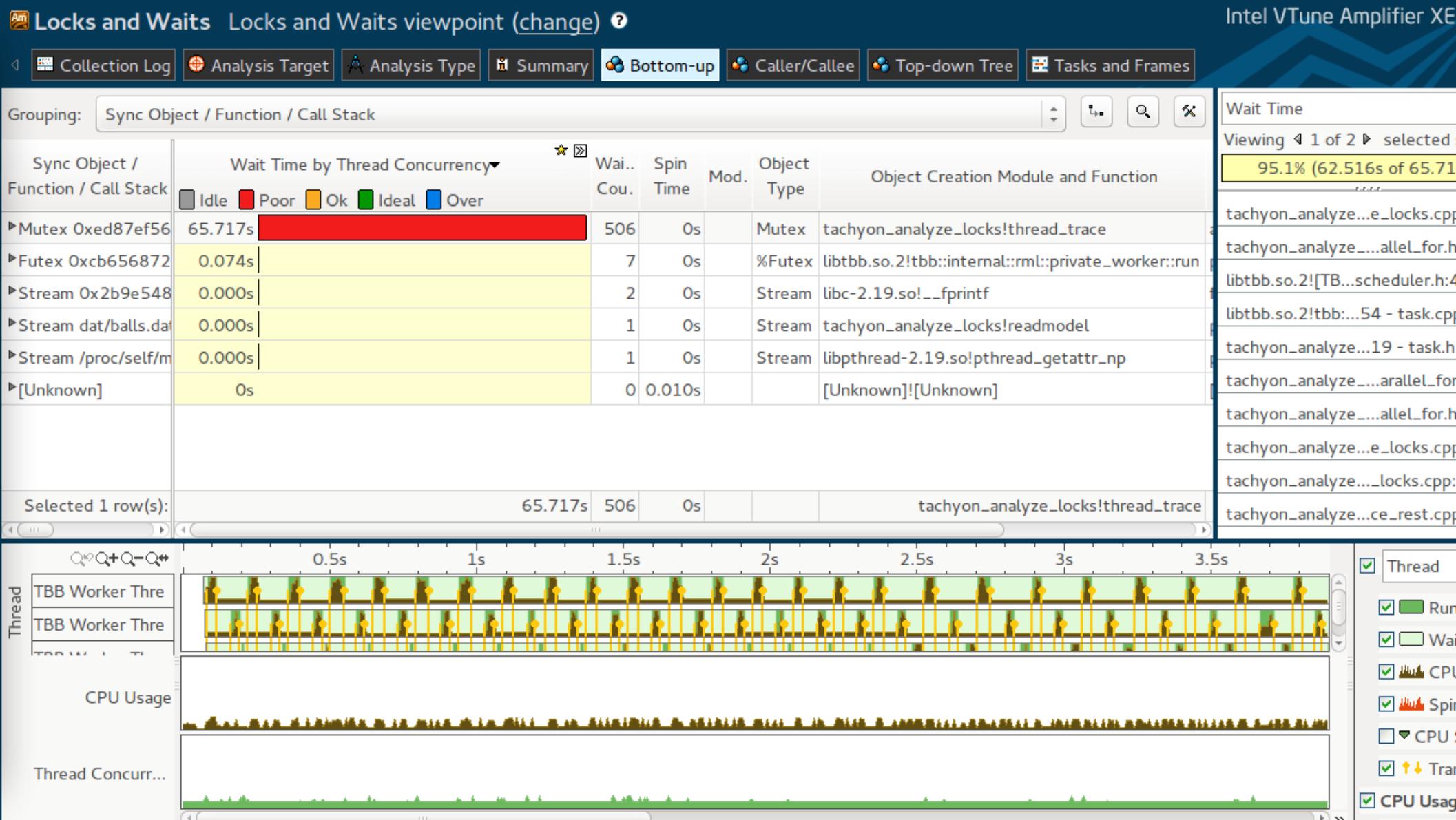
Selected 1 row(s): 106.977s

Hotspots



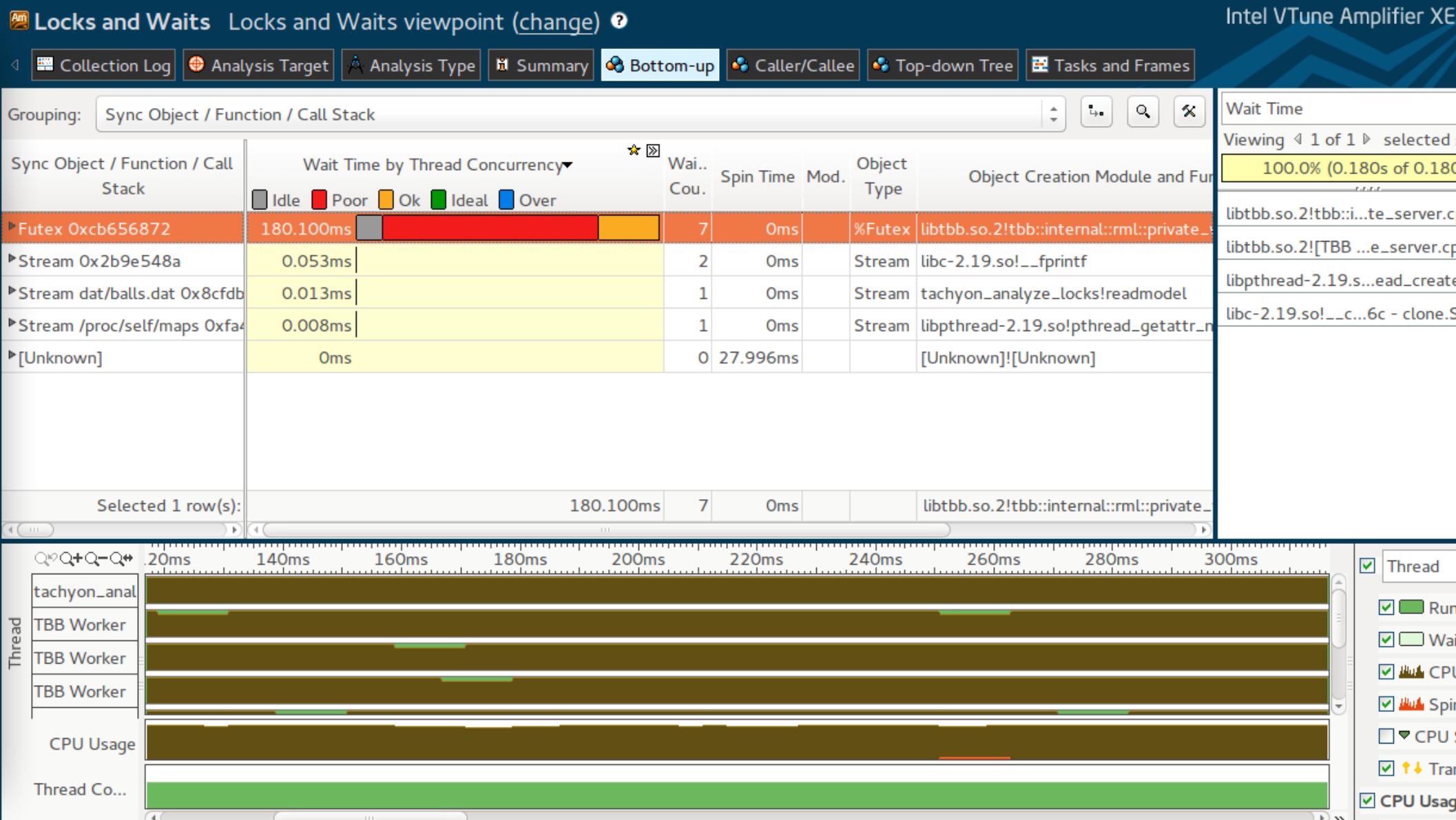
Bottlenecks

- ❖ Понякога в реализациите на даден паралелен алгоритъм се допускат (логически) грешки;
- ❖ Съществуват множество инструменти за анализ на работата и производителността на програмите, включително такива, които ни дават възможност да открием проблеми при паралелни програми;
- ❖ На следващият слайд е показан анализа на една примерна програма (Ray tracer-a Tachyon), в която има допусната грешка (използвана е критична секция с прекалено голяма общност на заключване – заключва се целия буфер на резултатата при запис в него);



Bottlenecks

- ❖ На графиката ясно се вижда, че отделните процеси се изчкват един друг и на практика в даден момент работи само едно ядро на многоядрения процесор;
- ❖ Вижда се и къде е изразходвано най-много време за “чакане” и от къде в програмата идва проблема;
- ❖ На практика буфера се е общ ресурс и е добре да се заключва при ползване, но алгоритъма е така реализиран, че две нишки никога не пишат в един и същи пиксел в него, така че в случая заключването е напълно излишно. Дори и да има запис в един и същ пиксел то е добре да заключваме само него, а не целия буфер;
- ❖ След премаване на излишното заключване полуаваме:



Неблокиращи Алгоритми и Структури Данни

Блокиране

- ❖ Традиционният подход за програмирането с много нишки е използването на примитиви за синхронизиране на достъпа до споделени ресурси;
- ❖ Примитиви за синхронизация като мутекси, семафори и критични секции са всички механизми, чрез които програмистът може да гарантира, че определени секции от код не се изпълняват едновременно, ако това би “повредило” споделените структури на паметта;
- ❖ Ако една нишка се опита да влезе в критична секция, в която се намира друга нишка, това води до блокиране на опитващата се да влезе в критичната секция, докато другата не я напусне;

Избягване на блокирането

- ❖ В някои случаи това може да се избегне;
- ❖ Използват се атомарни Read-Modify-Write операции;

Четене-Копиране-Обновяване и др.

- ❖ Zero-Copy – Подход, който се използва не само при паралелните приложения, свързан с избягването от копиране на памет (особено когато става въпрос за голями обеми данни). При него един процес предава данните и собственоста върху тях (“по указател”) на друг без необходимост от заделяне на нови буфери и трансфер на данните от единия буфер в другия;
- ❖ RCU – механизъм, чрез който може да се избегне необходимостта от заключване при паралелна работа и достъп до общи структури от данни. Използва се в ядрото на Linux и други ОС, във базите от данни под формата на версионизиране на записисте и др;

Четене-Копиране-Обновяване и др.

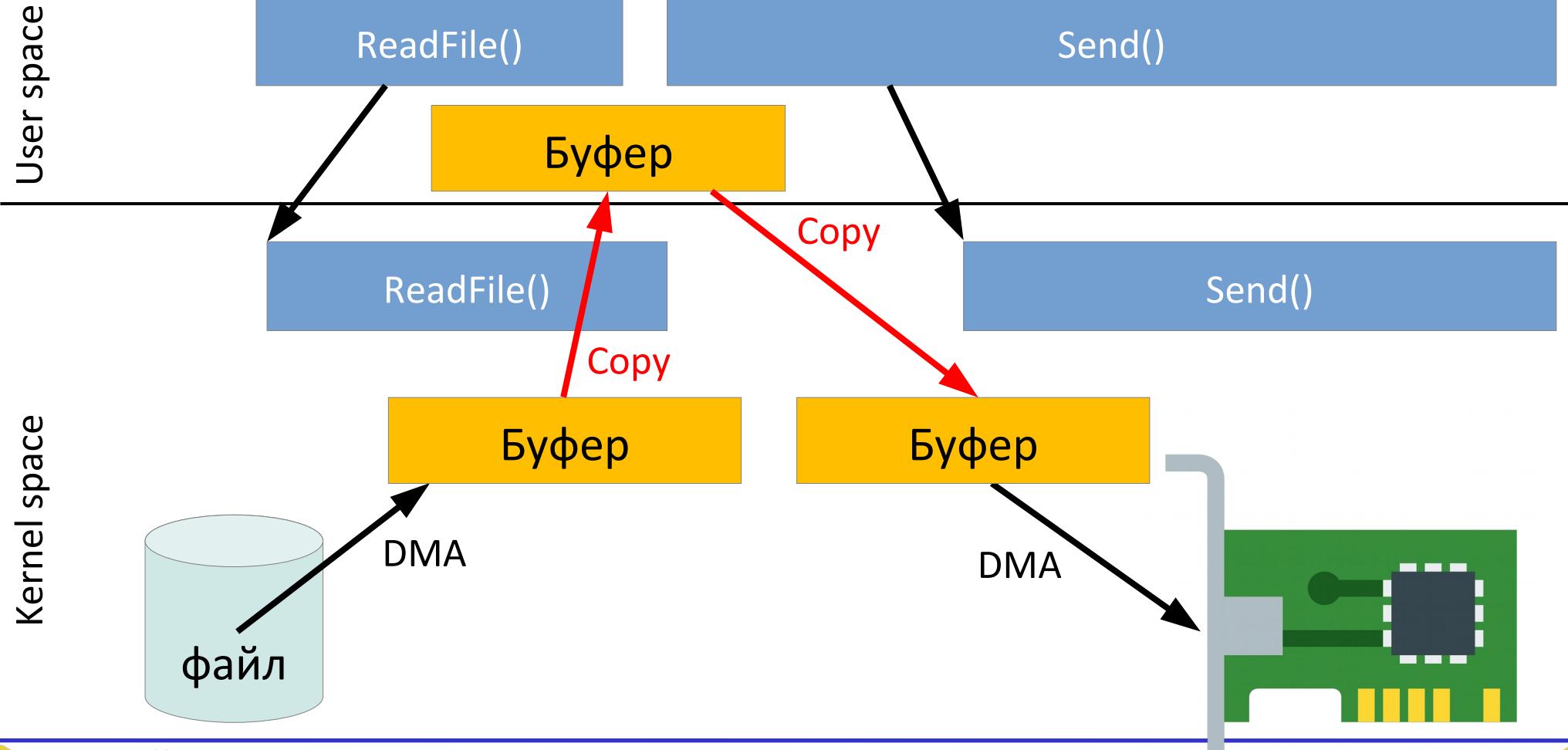
- ❖ Транзакционна памет – в съвременните процесори започва да се включват хардуерно ускорена поддръжка на тази концепция. Това позволява работата на паралелните програми, транзакциите в базите данни и др. да се ускорят многократно;
- ❖ Неблокиращи алгоритми и структури от данни;

Zero-Copy

Zero-Copy

- ❖ Това са компютърни операции, при които процесорът не изпълнява задачата да копира данни от една област памет в друга;
- ❖ Това често се използва за спестяване на цикли на процесора и честотна лента на паметта (например, при предаване на файл по мрежа, той се чете и изпръща с използването на един буфер, а не се копира от ОС в буфер на програмата, след което този буфер да се копира в ОС за да бъде предаван по мрежата – традиционното предаване на файл по мрежата изисква поне две копирания на данните, както и поне две превключвания на контекста между kernel и user пространството);
- ❖ Съвременните HSA архитектури правят възможно Zero-Copy между CPU и GPU;

Send File – класически подход



SendFile

User space

SendFile()

SendFile()

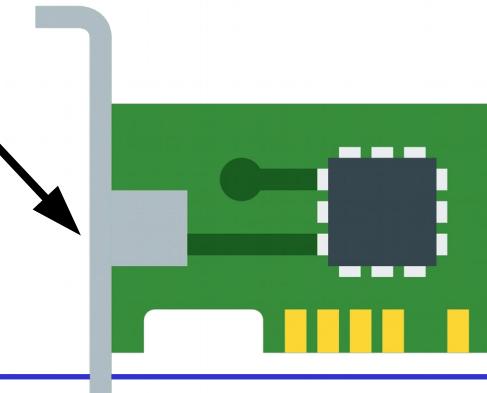
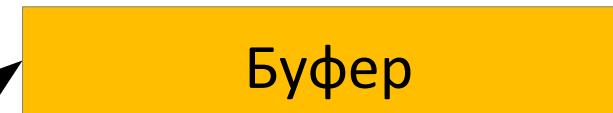
Kernel space

Буфер



DMA

DMA



SendFile – async

User space

async SendFile()

Sent()

SendFile()

Finish

Kernel space



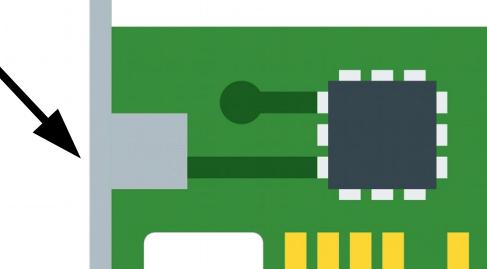
файл

DMA

Буфер

A yellow rectangular box with black text, representing a buffer.

DMA



Read-Copy-Update (RCU)

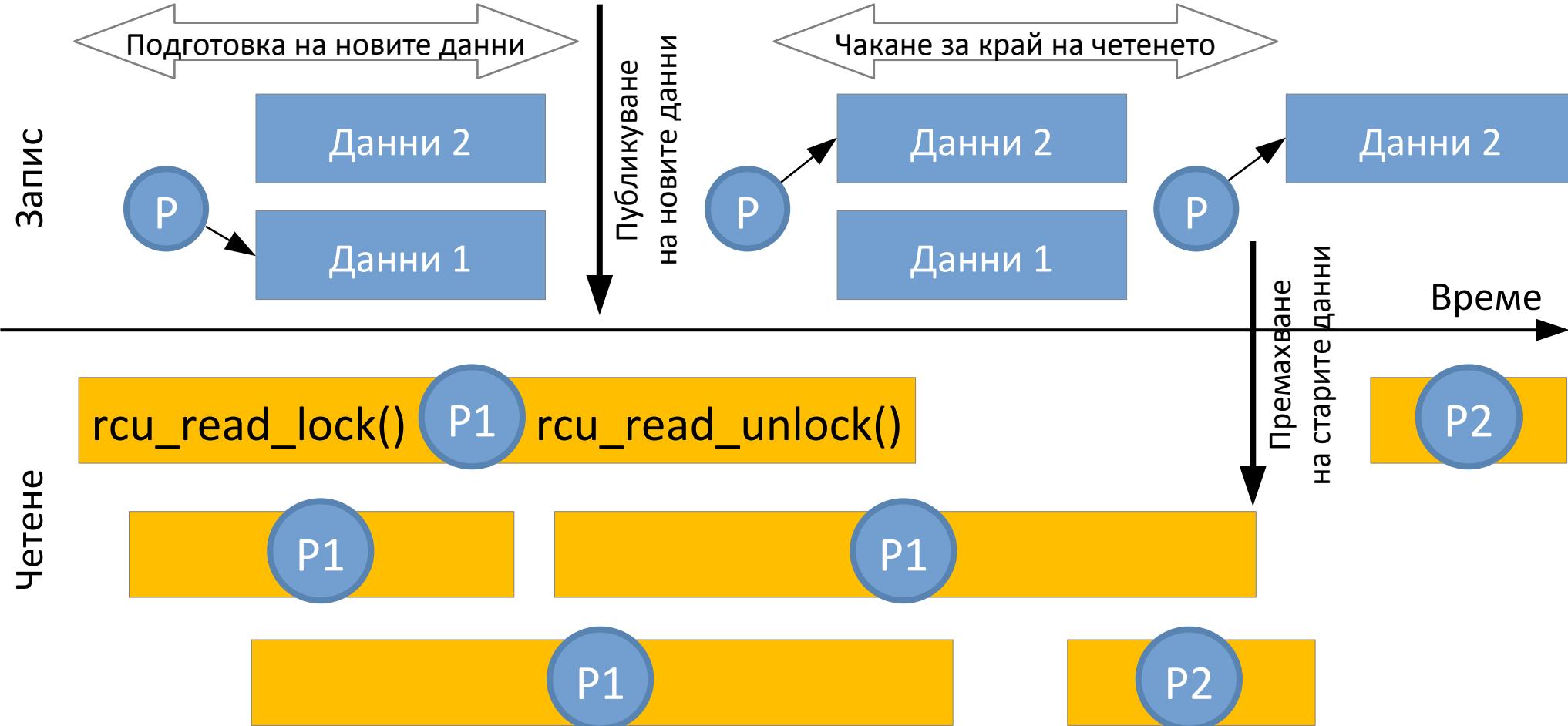
Copy-On-Write (COW)

и др.

- ❖ Read-copy-update (RCU) е механизъм за синхронизация, основан на взаимно изключване, но RCU не прилага взаимно изключване в конвенционалния смисъл: RCU четците могат и работят едновременно с RCU актуализации. Вариантът на взаимно изключване на RCU е в пространството, като четците на RCU имат достъп до стари версии на данните, които се актуализират едновременно, а не във времето, както е при конвенционалните механизми за контрол на паралелността;
- ❖ Използва се, когато изпълнението на четенията е от решаващо значение. Този подход е компромис между памет и време, позволяващ бързи операции с цената на повече заета памет;

- ❖ Read-copy-update позволява на множество нишки ефективно да се четат от споделена памет чрез отлагане на актуализациите за предварително съществуващите четения за по-късно време, като същевременно новите читатели ще прочетат актуализираните данни;
- ❖ Това кара всички читатели да продължат работата си така, сякаш няма синхронизация, следователно четенията ще бъдат бързи, но също така актуализациите ще бъдат затруднени;

RCU протокол



Транзакционна памет

(Transactional Memory)

Транзакционна памет

- ❖ Има два вида ТМ – софтуерна (STM) и хардуерна (ТМ подпомогната от специални инструкции в процесора);
- ❖ STM често е много оптимистичен подход: една нишка извършва модификации на споделената памет, без да се съобразява какво могат да правят другите нишки, записвайки всяко четене и запис в лог;
- ❖ Вместо да се натовари записващия с отговорността да се увери, че записа не се отразява неблагоприятно върху други операции в ход, отговорността е на “читателя”, който след приключване на цяла транзакция проверява дали други нишки не са направили едновременно промени в паметта (и затова прочетеното вече не е актуално);

Транзакционна памет

- ❖ Тази последна операция, при която промените на транзакция се валидират и ако валидирането е успешно, тогава промените стават постоянни се нарича “commit”;
- ❖ Една транзакция може също да се прекрати по всяко време, отменяйки всички нейни досегашни промени;
- ❖ Ако транзакция не може да бъде завършена поради промени, тя обикновено се прекъсва и се изпълнява от самото начало, докато успее;



Паралелно Програмиране

Езици и Библиотеки (API) за
паралелно програмиране

доц. д-р Александър Пенев

Паралелност на различните нива на системата

- ❖ На ниво Език за програмиране:
 - ❖ Канали (Channel);
 - ❖ Съпрограми (Coroutines);
 - ❖ Фючърси и „Обещания“ (Futures and Promises);
- ❖ На ниво ОС:
 - ❖ Многозадачност – кооперативна многозадачност и превантивна (preemptive) многозадачност;
 - ❖ Време деление, което заменя последователната „пакетна обработка“ на задачите с едновременна употреба на система
 - ❖ Процеси (Process);
 - ❖ Нишки (Thread);
- ❖ На ниво хардуер и мрежа: по начало са паралелни (различни у-ва);

*Езици за
Паралелно Програмиране
(ЕПП)*



Езици с елементи на ПП

- ❖ Ada – ключова дума task и др. вградени в езика;
- ❖ Cilk, Cilk++, Cilk Plus – С и С++ базирани езици, в които чрез допълнителни ключови думи се дефинира паралелизъм (Fork-Join концепцията);
- ❖ Cω – С Omega е изследователски език, разширяващ възможностите на C# с асинхронни комуникации;
- ❖ Clojure – диалект на Lisp, изпълняван на Java VM. Силно функционален, поддържа STM и др.;
- ❖ Co-array Fortran;
- ❖ Concurrent Pascal;
- ❖ Eiffel;
- ❖ Elixir – функционален език. Конкурентност на базата на леки нишки, които могат да комуникират чрез съобщения;
- ❖ Emerald – използва нишки и монитори;

Езици с елементи на ПП

- ❖ Erlang – използва асинхронни предавания на съобщения и „nothing shared” подход;
- ❖ Gambit Scheme – функционален. Прилага мощен и прост модел на предаване на съобщения, както и вдъхновен от Erlang модел на конкурентност;
- ❖ Go – CSP модел, goroutines (вариант на съпрограми), асинхронен, channels, и др.;
- ❖ Java;
- ❖ Julia;
- ❖ Occam – Базиран на Communicating Sequential Processes (CSP);
- ❖ Occam- π – модерен вариант на occam, като са добавени и идеи от π -calculus;

Езици с елементи на ПП

- ❖ Orc – недетерминиран, разпределен и конкурентен език;
- ❖ P;
- ❖ Pict – π -calculus базиан;
- ❖ Rust – паралелизъм базиран на Send, Sync и Thread;
- ❖ Scala – реализира Erlang-стил actors върху JVM;
- ❖ SequenceL – чисто функционален, автоматично паралелизиращ;
- ❖ Unified Parallel C;
- ❖ Modula-2 – наследник на Pascal. Има поддръжка на coroutines;
- ❖ Modula-3 – поддържа threads, mutexes, condition variables;
- ❖ SuperPascal – базиран на Concurrent Pascal и Joyce;
- ❖ VHSIC Hardware Description Language (VHDL) – IEEE STD-1076;
- ❖ ...

Пример Occam

```
-- occam

PROC write.string(CHAN output, VALUE string[])=
    SEQ character.number = [1 FOR string[BYTE 0]]
        output ! string[BYTE character.number]

write.string(terminal.screen, "Hello World!")

var ch
PAR
    terminal.keyboard ? ch
    terminal.screen ! "."

```

Библиотеки за Паралелно Програмиране (API)

Библиотеки (API) за ПП

- ❖ OpenMP;
- ❖ MPI, MPI-2;
- ❖ Java concurrency framework;
- ❖ Task Parallel Library for .NET;
- ❖ Threading Building Blocks (TBB);
- ❖ CUDA;
- ❖ OpenCL;
- ❖ OpenHMPP;
- ❖ Apache Hadoop;
- ❖ Apache Spark;
- ❖ Apache Flink;
- ❖ Apache Beam;
- ❖ ...

Примери и Често Използвани ЕПП и API

C++11

(Futures, Promises, ...)

Фючърси (Futures) C++11

- ❖ Atomic;
- ❖ Thread;
- ❖ Mutex;
- ❖ Condition variable;
- ❖ Future & Promises;
- ❖ ...

Фючърси (Futures) C++11

```
#include <iostream>           // std::cout
#include <future>            // std::async, std::future
#include <chrono>             // std::chrono::milliseconds

// a non-optimized way of checking for prime numbers:
bool is_prime(int x) {
    for (int i=2; i<x; ++i) if (x%i==0) return false;
    return true;
}
```

Фючърси (Futures) C++11

```
int main() {
    // call function asynchronously:
    std::future<bool> fut = std::async(is_prime, 444444443);

    // do something while waiting for function to set future:
    std::cout << "checking, please wait";
    std::chrono::milliseconds span(100);
    while (fut.wait_for(span)==std::future_status::timeout)
        std::cout << '.' << std::flush;

    bool x = fut.get(); // retrieve return value

    std::cout << "\n444444443 " << (x?"is ":"is not") << " prime.\n";

    return 0;
}
```



Обещания (Promises) C++11

```
#include <iostream>           // std::cout
#include <functional>         // std::ref
#include <thread>             // std::thread
#include <future>              // std::promise, std::future

void print_int(std::future<int>& fut) {
    int x = fut.get();
    std::cout << "value: " << x << '\n';
}

int main() {
    std::promise<int> prom;                      // create promise
    std::future<int> fut = prom.get_future(); // engag. with future
    std::thread th1(print_int, std::ref(fut)); // send future to thr.
    prom.set_value(10);                          // fulfill promise
    th1.join();                                // (synchronizes with getting the future)
    return 0;
}
```



C#

(Threads, Parallel.For, PLINQ, Tasks, Futures, ...)

- ❖ Threads (System.Threading);
- ❖ Task Parallel Library (TPL) – Tasks.Parallel, Parallel.For, Parallel.ForEach, Parallel.Invoke, ... (System.Threading.Tasks);
- ❖ PLINQ;
- ❖ Работа с Task обекти, представлящи асинхронни операции;
- ❖ Futures;
- ❖ async + await, yield;

Tasks

```
using System.Threading; using System.Threading.Tasks;
class Program {
    static void Main(string[] args) {
        Task<int[]> parent = new Task<int[]>(() => {
            var results = new int[3];
            new Task(() => {Thread.Sleep(15000); results[0] = 0;},
                    TaskCreationOptions.AttachedToParent).Start();
            new Task(() => results[1] = 1,
                    TaskCreationOptions.AttachedToParent).Start();
            new Task(() => results[2] = 2,
                    TaskCreationOptions.AttachedToParent).Start();
            return results;
        });
        // ...
    }
}
```

Tasks

```
// ...

parent.Start();
var finalTask = parent.ContinueWith(
    parentTask => {
        foreach (int i in parentTask.Result)
            Console.WriteLine(i);
    });
finalTask.Wait();
}

}
```

Parallel.For

```
using System.Linq;
using System.Threading; using System.Threading.Tasks;

class Program {
    static void Main(string[] args) {
        Parallel.For(0, 10, i => {
            Thread.Sleep(1000);
        });
        var numbers = Enumerable.Range(0, 10);
        Parallel.ForEach(numbers, i => {
            Thread.Sleep(1000);
        });
    }
}
```



PLINQ

```
var source = Enumerable.Range(1, 10000);

// Opt in to PLINQ with AsParallel.
var evenNums = from num in source.AsParallel()
                where num % 2 == 0
                select num;

Console.WriteLine("{0} even numbers out of {1} total",
                  evenNums.Count(), source.Count());
```

Резултат:

5000 even numbers out of 10000 total

Java

(Threads, Futures, Streams, ...)

Поддръжка на паралелни примитиви в езика и във framework-а

- ❖ Синхронизирани методи (като част от ЕП – ключова дума);
- ❖ Threads (java.lang.Thread и java.lang.Runnable);
- ❖ Futures, (java.util.concurrent);
- ❖ Atomic и Locks (java.util.concurrent.atomic, java.util.concurrent.locks);
- ❖ Streams (java.util.stream);

Синхронизирани методи (*lock*)

```
public synchronized void critical() {  
    // some thread critical stuff here  
}  
  
public void add(String site) {  
    synchronized (this) {  
        if (!crawledSites.contains(site)) {  
            linkedSites.add(site);  
        }  
    }  
}
```

Thread

```
class PrimeThread extends Thread {  
    long minPrime;  
  
    PrimeThread(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime ...  
    }  
}  
  
// Create and start thread  
PrimeThread p = new PrimeThread(143);  
p.start()
```



Runnable

```
class PrimeRun implements Runnable {  
    long minPrime;  
    PrimeRun(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime ...  
    }  
}  
  
// Create a thread and start it running  
PrimeRun p = new PrimeRun(143);  
new Thread(p).start();
```



Locks

```
class X {  
    private final ReentrantLock lock = new ReentrantLock();  
  
    public void m() {  
        lock.lock(); // block until condition holds  
        try {  
            // ... method body  
        } finally {  
            lock.unlock()  
        }  
    }  
}
```

Брояч без Atomics

```
public class SafeCounterWithLock {  
    private volatile int counter;  
  
    public synchronized void increment() {  
        counter++;  
    }  
}
```

Брояч с *Atomics*

```
public class SafeCounterWithoutLock {  
    private final AtomicInteger counter = new AtomicInteger();  
  
    public void increment() {  
        while(true) {  
            int existingValue = counter.getValue();  
            int newValue = existingValue + 1;  
            if(counter.compareAndSet(existingValue, newValue)) {  
                return;  
            }  
        }  
    }  
}
```

Streams

```
List<String> memberNames = new ArrayList<>();  
memberNames.add("Amitabh");  
memberNames.add("Shekhar");  
memberNames.add("Aman");  
memberNames.add("Rahul");  
memberNames.add("Shahrukh");  
memberNames.add("Salman");  
memberNames.add("Lokesh");  
  
memberNames.stream().filter((s) -> s.startsWith("A"))  
    .forEach(System.out::println);
```

Резултат:

Amitabh

Aman



Streams

```
memberNames.stream().filter((s) -> s.startsWith("A"))
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

Резултат:

AMITABH

AMAN

Streams

```
memberNames.stream().sorted()  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

Резултат:

AMAN

AMITABH

LOKESH

RAHUL

SALMAN

SHAHRUKH

SHEKHAR

Streams

```
List<String> memNamesInUppercase =  
memberNames.stream().sorted()  
    .map(String::toUpperCase)  
    .collect(Collectors.toList());  
  
System.out.print(memNamesInUppercase);
```

Резултат:

[AMAN, AMITABH, LOKESH, RAHUL, SALMAN, SHAHRUKH, SHEKHAR]

Streams

```
Optional<String> reduced = memberNames.stream()
    .reduce((s1,s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
```

Резултат:

Amitabh#Shekhar#Aman#Rahul#Shahrukh#Salman#Lokesh

Streams

```
public class StreamBuilders {  
    public static void main(String[] args){  
        List<Integer> list = new ArrayList<Integer>();  
        for(int i = 1; i < 10; i++){  
            list.add(i);  
        }  
        // Here creating a parallel stream  
        Stream<Integer> stream = list.parallelStream();  
        Integer[] evenNumbersArr = stream  
            .filter(i -> i%2 == 0).toArray(Integer[]::new);  
        System.out.print(evenNumbersArr);  
    }  
}
```

OpenMP

(Fork-Join, #pragma, ...)

Базиран основно Fork-Join

- ❖ Използва #pragma директиви на компилатора, за да “подскаже” на компилатора как искаме при възможност да се разпаралели програмата;
- ❖ Ако компилатора не поддържа OpenMP или не са зададени правилните параметри на компилатора, то директивата се игнорира и програмата се компилира както нормално;
- ❖ Shared memory многозадачен модел;
- ❖ Базиран на Fork-Join модела;
- ❖ Могат да се задават бариери и много други с параметрите на директивата #pragma omp ...

OpenMP пример

```
void simple(int n, float *a, float *b) {
    int i;

    #pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

MPI

(Комуникация, Синхронизация, ...)

MPI

- ❖ Комуникация (`MPI_Send`, `MPI_Recv`, `MPI_Sendrecv` и др.);
- ❖ Разпределяне на работата;
- ❖ Глобални редукции (`MPI_Reduce`, `MPI_Op_create`, `MPI_Allreduce`,
`MPI_Reduce_scatter`, `MPI_Scan` и др.);
- ❖ Синхронизация (`MPI_Barrier`, синхронни комуникации и др.);
- ❖ Дефиниране на потребителски типове;
- ❖ Дефиниране на виртуални топологии;
- ❖ Паралелен В/И (`MPI-2`);
- ❖ Стартiranе на процеси (`MPI-2`)
- ❖ И др.;

MPI – пример (Hello World)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment
    int world_size; // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int world_rank; // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    char processor_name[MPI_MAX_PROCESSOR_NAME]; // Name
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    printf("Hello world from proc %s, rank %d out of %d procs\n",
           processor_name, world_rank, world_size);

    MPI_Finalize(); // Finalize the MPI environment.
}
```



MPI – пример (Hello World)

host_file:

host1name
host2name
host3name
host4name

```
> export MPIRUN=/.../mpirun
> export MPI_HOSTS=host_file
> mpirun -n 4 -f host_file ./mpi_hello_world
Hello world from proc host2name, rank 1 out of 4 procs
Hello world from proc host1name, rank 0 out of 4 procs
Hello world from proc host4name, rank 3 out of 4 procs
Hello world from proc host3name, rank 2 out of 4 procs
```

MPI – пример (Редукции, SUM и AVG)

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
    local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);
```

MPI – пример (Редукции, SUM и AVG)

```
// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
    printf("Total sum = %f, avg = %f\n", global_sum,
           global_sum / (world_size * num_elements_per_proc));
}
```

MPI – пример (Редукции, SUM и AVG)

```
> mpirun -n 4 ./reduce_avg 100
```

```
Local sum for process 0 - 51.385098, avg = 0.513851
Local sum for process 1 - 51.842468, avg = 0.518425
Local sum for process 2 - 49.684948, avg = 0.496849
Local sum for process 3 - 47.527420, avg = 0.475274
Total sum = 200.439941, avg = 0.501100
```

POSIX Threads, Boost.Thread, TBB, ...

(Нишки, Синхронизация, ...)

Поддържат

- ❖ Поддържат създаване и управление на Нишки (Threads);
- ❖ Fork-Join;
- ❖ Mutexes;
- ❖ Condition variables;
- ❖ Синхронизация, read/write locks и бариери;
- ❖ Различни готови за използване „паралелни“ базови структури от данни;
- ❖ И др.;

pthreads – пример

```
#include <pthread.h>
#include <stdio.h>

/* this function is run by the second thread */
void *inc_x(void *x_void_ptr) {
    /* increment x to 100 */
    int *x_ptr = (int *)x_void_ptr;
    while(++(*x_ptr) < 100);

    printf("x increment finished\n");

    /* the function must return something - NULL will do */
    return NULL;
}
```



pthreads – пример

```
int main() {
    int x = 0, y = 0;

    /* show the initial values of x and y */
    printf("x: %d, y: %d\n", x, y);

    /* this variable is our reference to the second thread */
    pthread_t inc_x_thread;

    /* create a second thread which executes inc_x(&x) */
    if(pthread_create(&inc_x_thread, NULL, inc_x, &x)) {
        fprintf(stderr, "Error creating thread\n"); return 1;
    }
}
```



pthreads – пример

```
/* increment y to 100 in the first thread */
while (++y < 100);

printf("y increment finished\n");

/* wait for the second thread to finish */
if(pthread_join(inc_x_thread, NULL)) {
    fprintf(stderr, "Error joining thread\n"); return 2;
}

/* results - x is now 100 thanks to the second thread */
printf("x: %d, y: %d\n", x, y);

return 0;
}
```



Boost – пример

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>
void wait(int seconds) { boost::this_thread::sleep_for(
                           boost::chrono::seconds{seconds}); }
void thread() {
    for (int i = 0; i < 5; ++i) {
        wait(1);
        std::cout << i << '\n';
    }
}
int main() {
    boost::thread t{thread};
    t.join();
}
```



TBB – пример

```
#include "tbb/task_group.h"
using namespace tbb;
int Fib(int n) {
    if (n<2) {
        return n;
    } else {
        int x, y;
        task_group g;
        g.run([&]{x=Fib(n-1);}); // spawn a task.
        g.run([&]{y=Fib(n-2);}); // spawn another task.
        g.wait(); // wait for both tasks to
                  // complete.
        return x+y;
    }
}
```



CUDA

(Compute Unified Device Architecture)

CUDA

- ❖ Това е платформа за паралелни изчисления и API създадена от Nvidia за работа и програмиране на техните GPGPU;
- ❖ Поддържа езиците C, C++ и Fortran;
- ❖ Платформата включва и множество силно оптимизирани библиотеки с математически и други функции като cuBLAS, cuFFT, cuRAND и др;
- ❖ Включени са и средства за създаване, дебъгiranе, анализ и оптимизация на приложения базирани на CUDA;
- ❖ Подобно на OpenCL работата се базира на създаване на kernel-и и работа с тях;



Пример – Hello, CUDA!

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b) {
    a[threadIdx.x] += b[threadIdx.x];
}

int main() {
    char a[N] = "Hello \0\0\0\0\0\0";
    int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    char *ad; int *bd;
    const int csize = N*sizeof(char);
    const int isize = N*sizeof(int);
    printf("%s", a);
```



Пример – Hello, CUDA!

```
cudaMalloc( (void**)&ad, csize ) ;
cudaMalloc( (void**)&bd, isize ) ;
cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice ) ;
cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice ) ;

dim3 dimBlock( blocksize, 1 ) ;
dim3 dimGrid( 1, 1 ) ;
hello<<<dimGrid, dimBlock>>>(ad, bd) ;
cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost ) ;
cudaFree( ad ) ;
cudaFree( bd ) ;

printf("%s\n", a) ;

return EXIT_SUCCESS ;
}
```



Пример 2

```
import pycuda.compiler as comp
import pycuda.driver as drv
import numpy
import pycuda.autoinit
mod = comp.SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)
multiply_them(drv.Out(dest), drv.In(a), drv.In(b), block=(400,1,1))
print dest-a*b
```

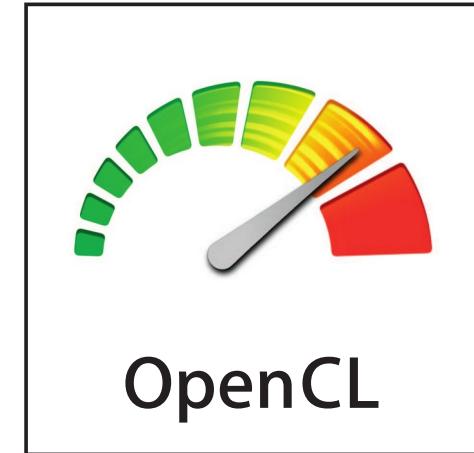


OpenCL

(Open Computing Language)

OpenCL

- ❖ Това е стандартна софтуерна рамка (framework) за писане на хетерогенни и силно паралелни приложения;
- ❖ Използват се езици базирани на C99 и C++11, разширени с ключови думи и примитиви за програмиране на широк клас устройства;
- ❖ Поддържа CPU, GPU, Digital signal processors (DSPs), Field-programmable gate arrays (FPGAs) и други процесори и хардуерни ускорители;
- ❖ Всяка програма се разделя на две части – базова (написана на host езика на основната програма) и kernel-и написани на OpenCL C или OpenCL C++, които се компилират от вградения в OpenCL библиотеката компилатор;



OpenCL пример – умножение на вектор и матрица (OpenCL C kernel)

```
// matvec.cl
__kernel void matvec(__global const float *A,
                      __global const float *x,
                      uint ncols, __global float *y)
{
    size_t i = get_global_id(0);           // Global id row index
    __global float const *a = &A[i*ncols]; // I'th row ptr
    float sum = 0.f;                     // For dot product
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

OpenCL пример – главна програма

```
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>
int main() {
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    auto platform = platforms.front();

    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_CPU, &devices);
    auto device = devices.front();

    std::ifstream demoFile("matvec.cl");
    std::string src(std::istreambuf_iterator<char>(demoFile),
                   (std::istreambuf_iterator<char>()));
}
```

```
cl::Program::Sources sources(1,
    std::make_pair(src.c_str(), src.length() + 1));

cl::Context context(device);
cl::Program program(context, sources);
auto err = program.build("-cl-std=CL1.2");

float A[16][3]; float x[3]; float y;
cl::Buffer memBufA(context, CL_MEM_HOST_READ_ONLY, sizeof(A));
cl::Kernel kernel(program, "matvec", &err);
kernel.setArg(0, memBufA); // Other params and result ...

cl::CommandQueue queue(context, device);
queue.enqueueWriteBuffer(memBufA, GL_TRUE, 0, sizeof(A), A);
queue.enqueueWriteBuffer(memBufX, GL_TRUE, 0, sizeof(x), x);
queue.enqueueTask(kernel);
queue.enqueueReadBuffer(memBufY, GL_TRUE, 0, sizeof(y), y);

cout << y;
```



OpenHMPP, OpenACC, C++ AMP

(Коделети, кернели, #pragma, GPU, ...)

HMPP (Hybrid Multicore Parallel Programming)

- ❖ Работа с хетерогенен хардуер;
- ❖ Базиран е на подход подобен на OpenMP с използване на `#pragma` анотации;
- ❖ Хетерогенните изчисления се програмират в т.нар. **коделети** – прости функции написани на базовия език (С или Fortran), които се компилират до езика на „target“ платформи като CUDA, OpenCL и др.;
- ❖ Може да се използва съвместно с OpenMP, MPI и др.;

OpenACC, C++ AMP, ...

- ❖ Аналогично OpenACC (open accelerators), предоставя подобен подход, но се поддържат C, C++ и Fortran. Коделетите тук се наричат **kernels**;
- ❖ Подобно е предназначението и подхода в C++ AMP. Първоначално той е разработка на Microsoft и се базира на превеждане на прости функции, ламбда изрази и други маркирани с ***restrict(amp)*** до извиквания на API функции базирани на DirectX 11. По-късно се появяват и реализации реализирани на базата на Clang/LLVM и OpenCL;

OpenHMPP – пример

```
#pragma hmpp simple1 codelet, args[outv].io=inout, target=CUDA
static void matvec(int sn, int sm, float inv[sm],
                   float inm[sn][sm], float *outv) {
    int i, j;
    for (i = 0 ; i < sm ; i++) {
        float temp = outv[i];
        for (j = 0 ; j < sn ; j++) {
            temp += inv[j] * inm[i][j];
        }
        outv[i] = temp;
    }
    int main(int argc, char **argv) {
        int n;
        /* codelet use */
        #pragma hmpp simple1 callsite, args[outv].size={n}
        matvec(n, m, myinc, inm, myoutv);
    }
```



Въпроси?

apenev@uni-plovdiv.bg



Паралелно Програмиране

Бъдеще на
Паралелните Архитектури и Програмиране

доц. д-р Александър Пенев

Риска на прогнозите

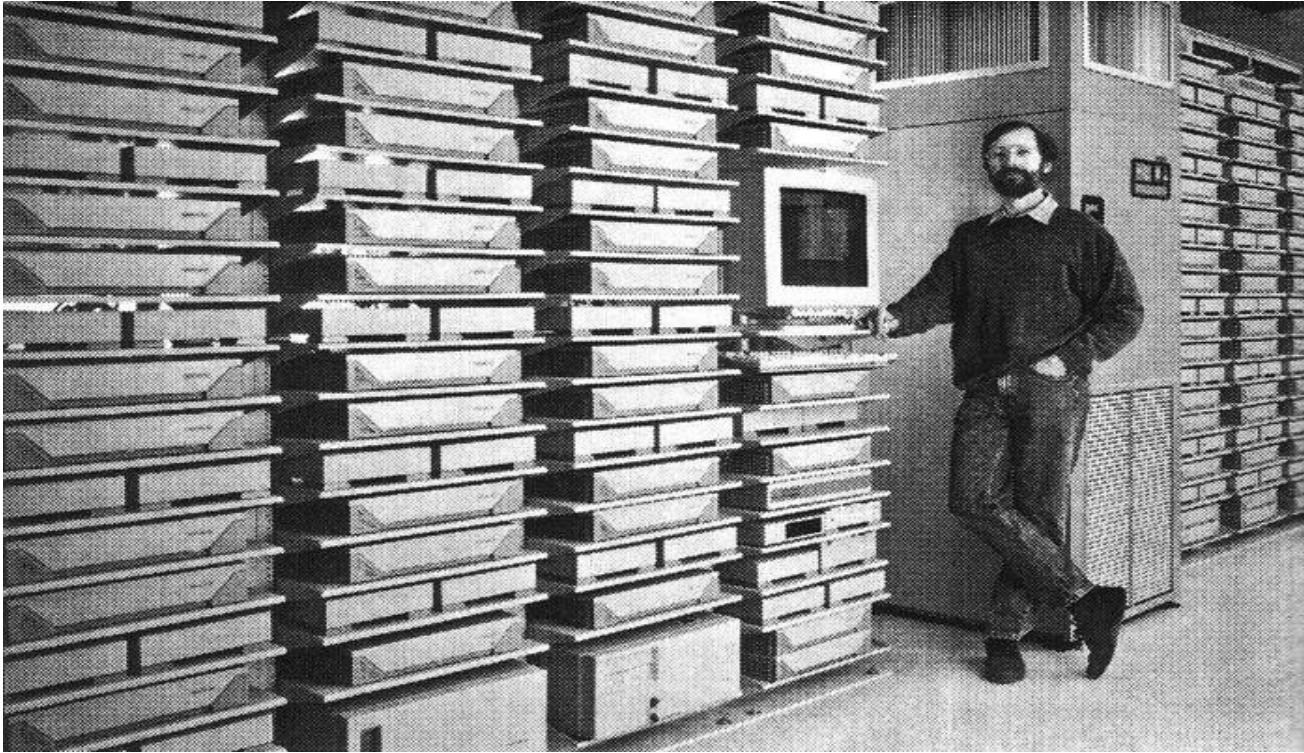
- ❖ Много е трудно човек да каже със сигурност какво ще стане утре;
- ❖ Какво ще стане в следващите 10-20 години е още по-трудно;
- ❖ Много хора с опит са правили прогнози за ИТ индустрията и дори само няколко години след това изказванията им звучат смешно;
- ❖ Понякога се появяват технологии, които “преобръщат” цялата ИТ индустрия. Например персоналните компютри;
- ❖ Въпреки това се наблюдават някои тенденции, които е силно вероятно да се запазят в близко бъдеще. Например „Законът на Мур“ все още не е „отменен“;

- ❖ Бъдещето е комбинация на Еволюция и Революция;

- ❖ „Аз мисля, че има световен пазар за може би 5 компютъра.” – Томас Уотсън, председател на IBM, 1949;
- ❖ „Няма причина в света всеки да иска да има компютър в дома си. Няма причина.” – Кен Олсън, председател на DEC, 1977;
- ❖ „640 RAM трябва да са достатъчни за всеки.” – Бил Гейтс, 1981;

Бъдеще на Паралелните Архитектури

GPGPU



Ed Catmull в рендеринг „фермата“ на Pixar, 1995



Кълстер от 117 (87 дву процесорни и 30 четири процесорни, 100 MHz) SPARCstation 20s със от 192 до 384 MB RAM, Всеки с 4-5GB дисково пространство.

Te работят със Solaris и софтуер Pixar's "Renderman" и SparcServer 1000e за разпределение на задачите.



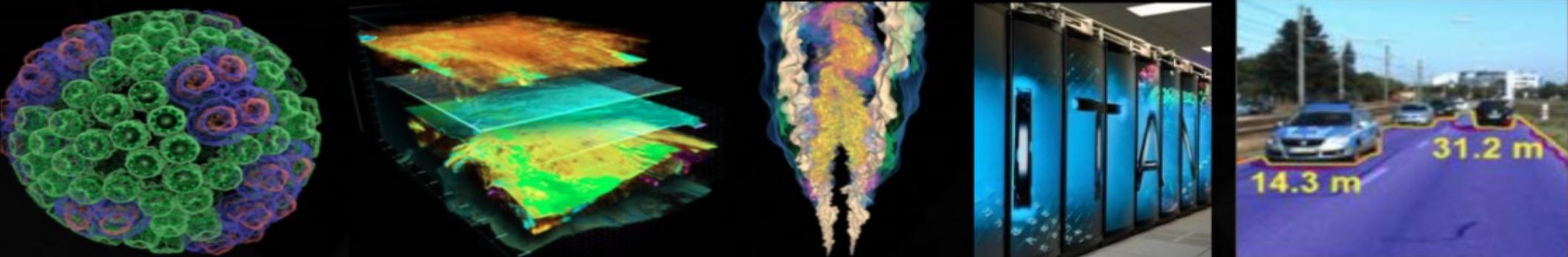
- ❖ Когато Pixar създават рендеринг фермата си през 1995-та и пускат на пазара филма „Играта на играчките“, графичните ускорители в персоналните компютри все още едва „прохождат“. Понятието за GPU, GPGPU и това че един ден супер-компютрите ще са базирани в голямата си част на GPGPU е фантастика;
- ❖ Nvidia току що е основана и малко хора са чували за нея. На пазара с „графика“ и „графични ускорители“ има други играчи, някои от които днес не съществуват;

GPGPU – история

A BRIEF HISTORY OF GPGPU



General-Purpose computation on Graphics Processing Units

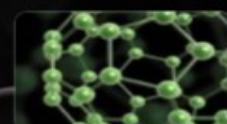


GPGPU – приложенията са далеч извън “графичните”

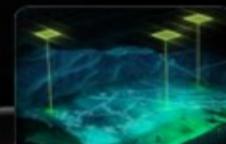
FROM HPC TO ENTERPRISE DATACENTERS



Oil & Gas



Higher Ed



Government



Supercomputing



Finance



Consumer Web

Schlumberger

HARVARD
School of Engineering
and Applied Sciences

STANFORD
UNIVERSITY

Raytheon



Naval Research
Laboratory

cscs

NCSA

Tokyo Institute
of Technology

OAK
RIDGE
National Laboratory

Lawrence Livermore
National Laboratory

J.P.Morgan

BARCLAYS

STANDARD LIFE

BNP PARIBAS

MUREX™

Baidu 百度

salesforce
SOFTBANK

SHAZAM

amazon.com

Yandex

PETROBRAS

Eni

Chevron

Statoil

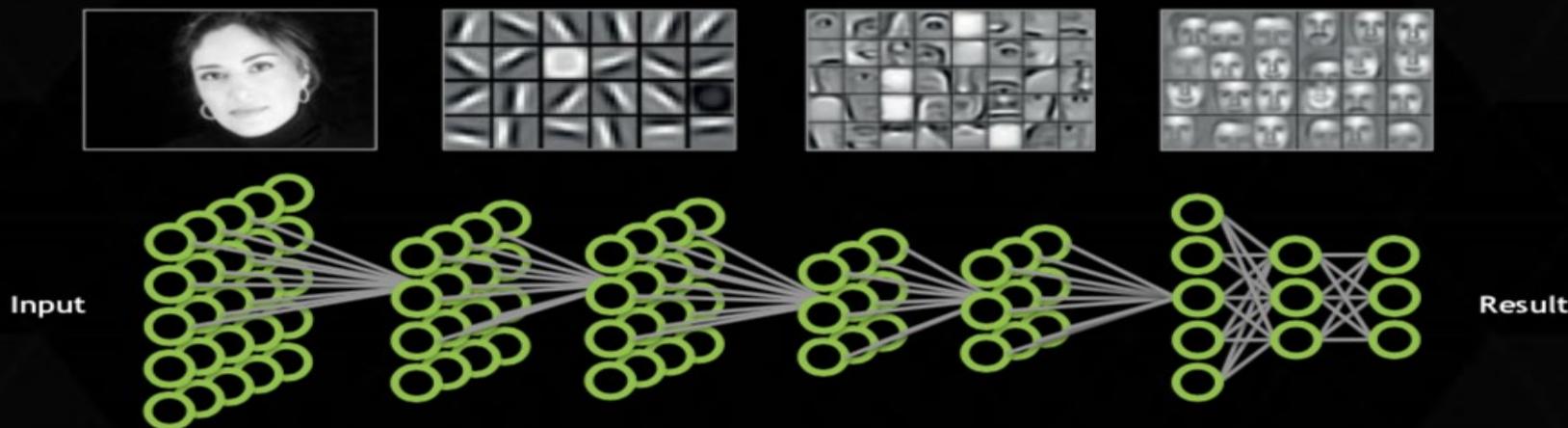
Georgia Tech

ETH
Swiss Federal Institute of Technology Zurich

UNIVERSITY OF CAMBRIDGE

GPGPU (DNN)

MACHINE LEARNING USING DEEP NEURAL NETWORKS



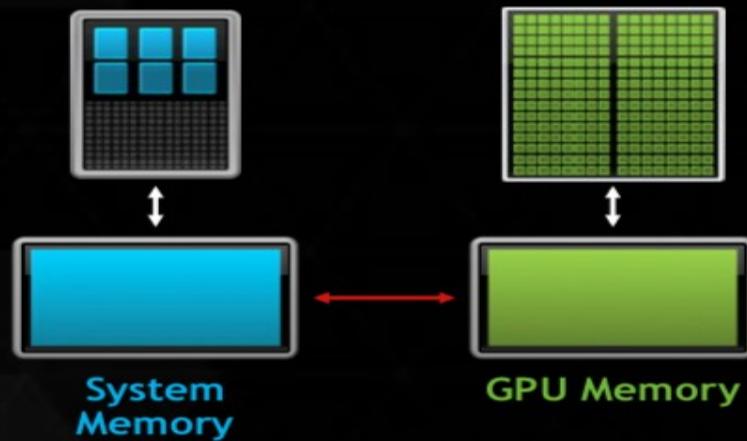
Sinton et al., 2006; Bengio et al., 2007; Bengio & LeCun, 2007; Lee et al., 2008; 2009

Visual Object Recognition Using Deep Convolutional Neural Networks

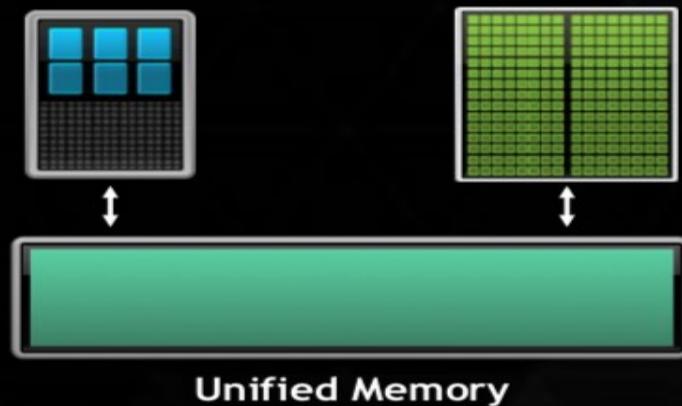
Rob Fergus (New York University / Facebook) <http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php#2985>

Unified Memory DRAMATICALLY LOWER DEVELOPER EFFORT

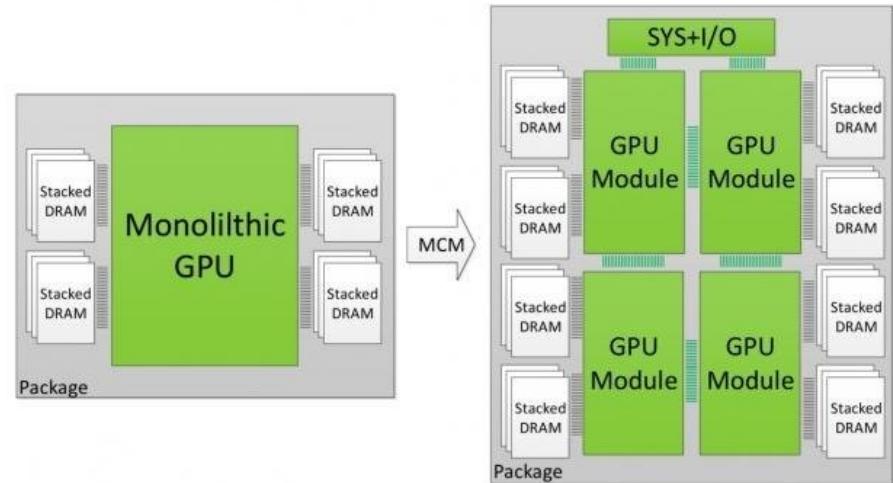
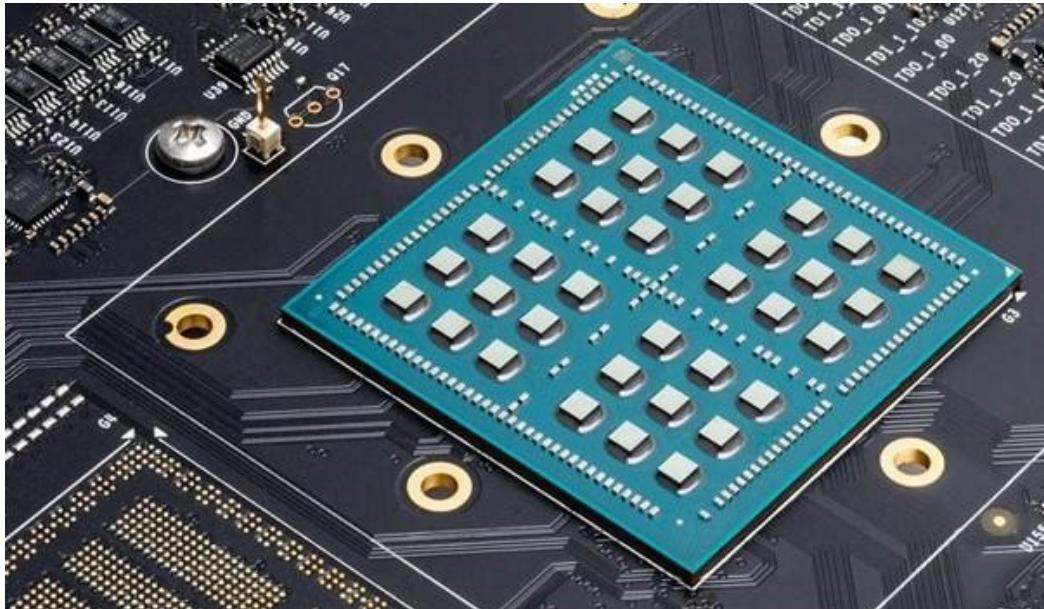
Past Developer View



Developer View With Unified Memory



NVIDIA Next Gen-GPU Hopper



Новата архитектура на
nVidia Hopper

Преход от монолитни GPU към модулна архитектура на GPU.

GEFORCE RTX



Игри с RTX



Minecraft – <https://youtu.be/UCfuZSEFzIg>

Игри с RTX

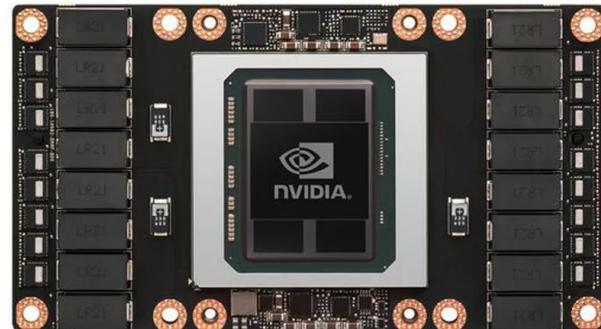


Battlefield V – <https://youtu.be/rpUm0N4Hsd8>

Развитие на GPGPU



Intel Phi

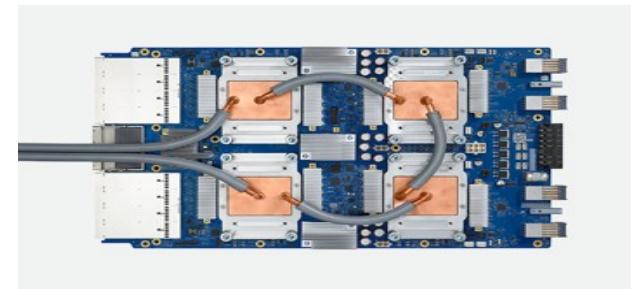


Nvidia Tesla



TPU

Доставчиците на Cloud услуги предлагат лесен достъп до специализирания хардуер (Tensor Processing Unit)

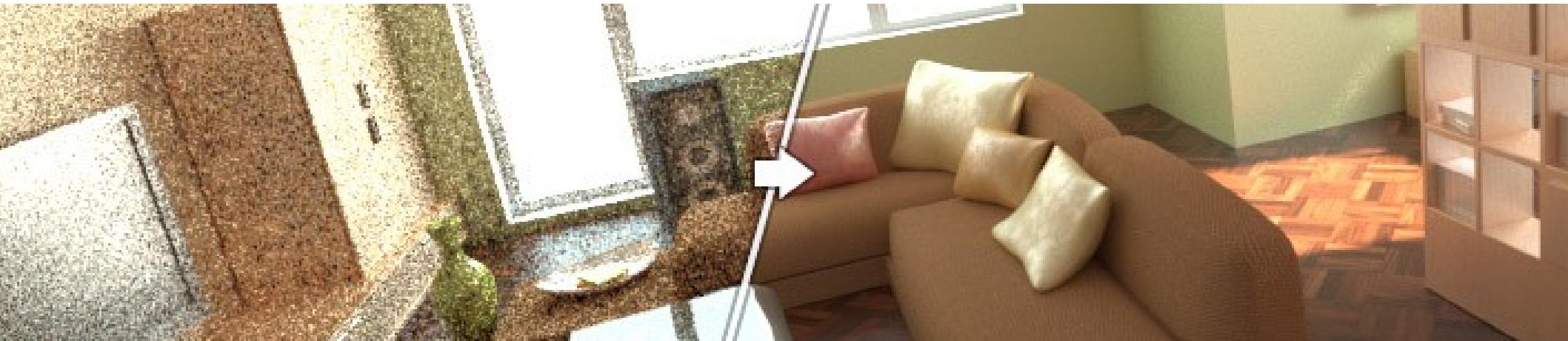


- TPU Card to replace a disk
- Up to 4 cards / server

TPU Card & Package

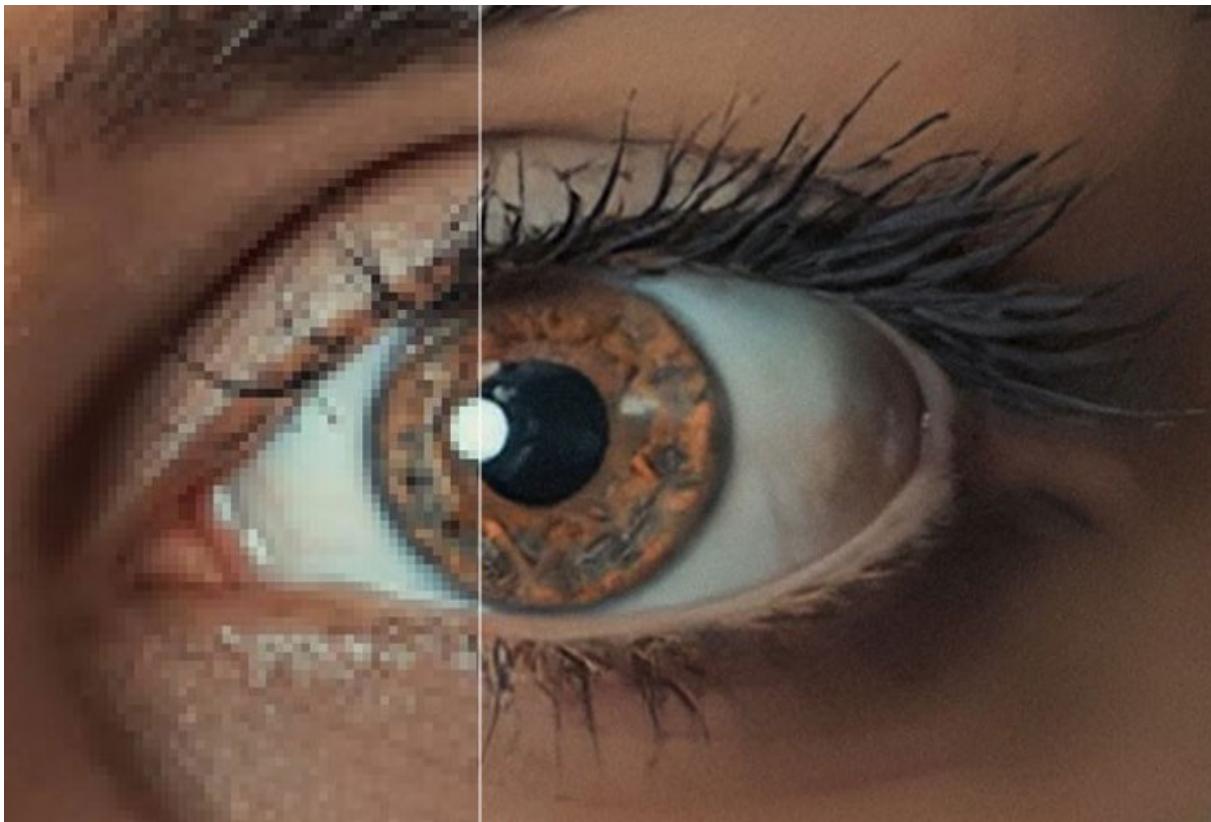


Denoising в реално време (с използване на CNN)

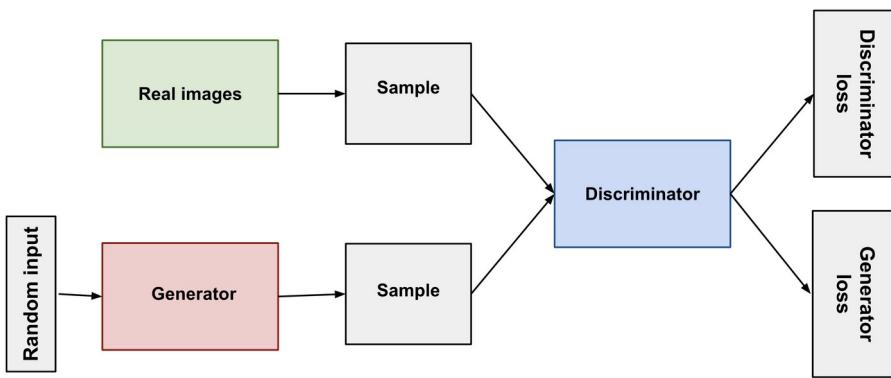


През 2017 nVidia представиха своя деноисер работещ
в реално време, базиран на невронни мрежи

Увеличаване на Изображения с AI



GAN – генеративни невронни мрежи



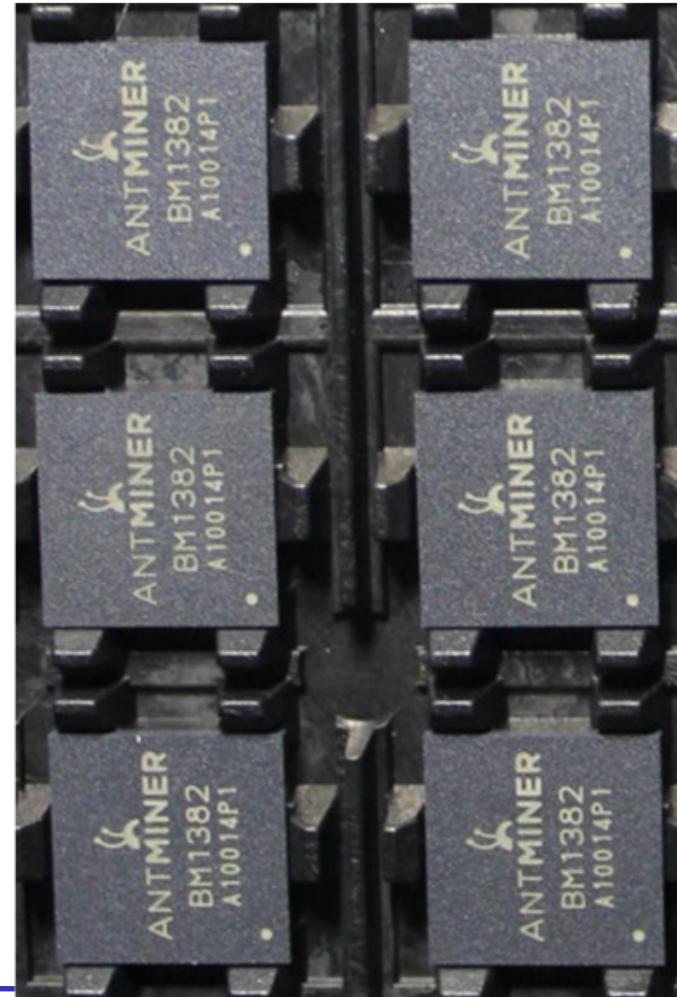
GAN – видео филтър “премахване на дъжд”



ASIC

ASIC – Application Specific Integrated Circuit

- ❖ Интегрални схеми с конкретно приложение;
- ❖ За разлика от CPU, GPU и др., които са домейн-специфични, ASIC са приложно-специфични;
- ❖ Дизайнът на такива „хардуерни приложения“ става с помощта на специализирани езици за „програмиране“, наричани още Hardware Description Language (HDL). Такива са Verilog, VHDL и др. Чрез тях се описва функционалността на ASIC. Тези езици са силно паралелни;



FPGA

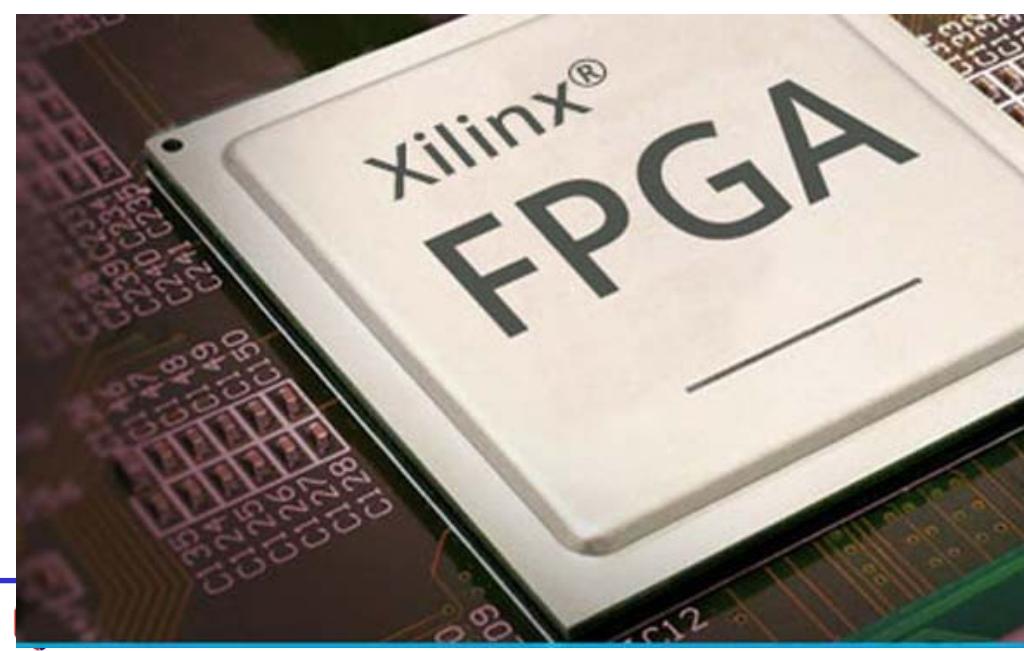
FPGA – Field-Programmable Gate Array

- ❖ Интегрирани схеми, проектирани да бъдат конфигурирани от клиента или дизайнера след производството;
- ❖ Конфигурацията на FPGA обикновено се специфицира с помощта на хардуерен език за описание (HDL), подобен на този, използван за ASIC;
- ❖ Съдържат множество логически блокове, които могат да бъдат конфигурирани и свързвани;
- ❖ За разлика от ASIC, FPGA могат да се препрограмират т.е. да им се правят хардуерни ъпдейти;
- ❖ Недостатък е че са по-бавни и използват повече енергия;





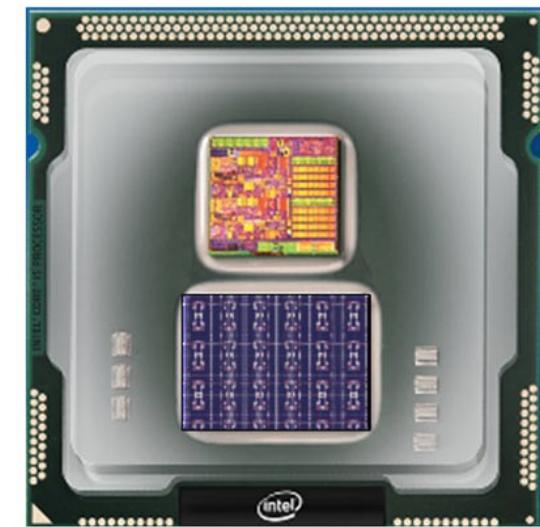
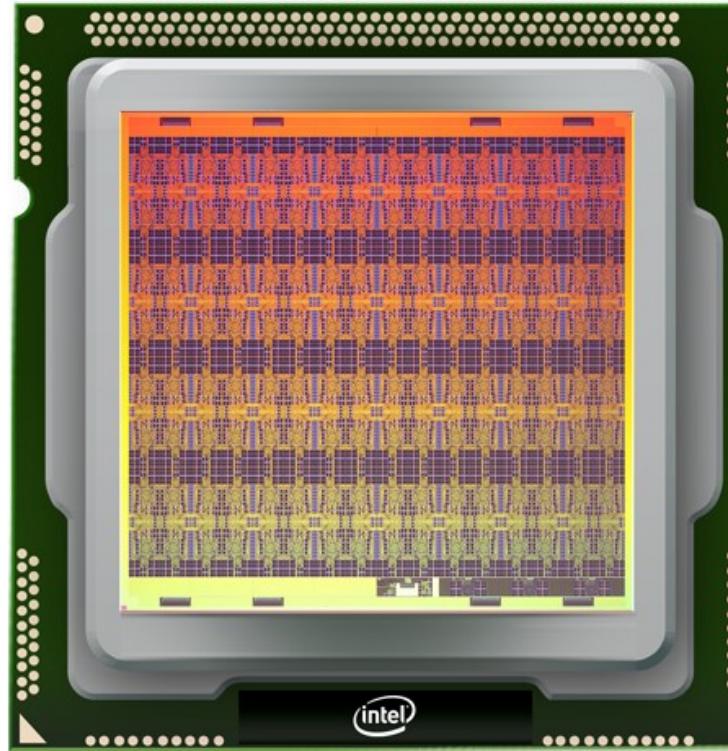
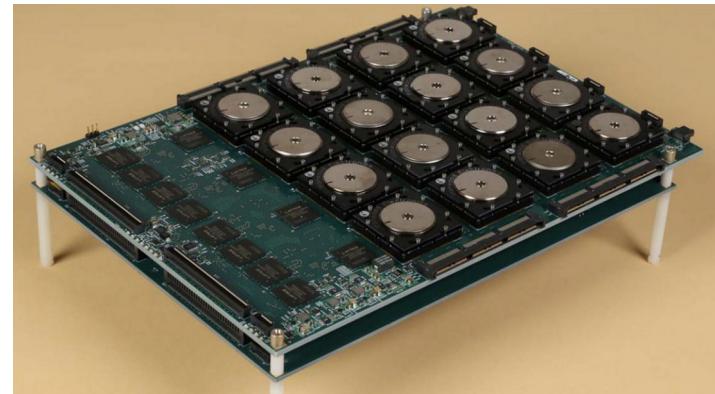
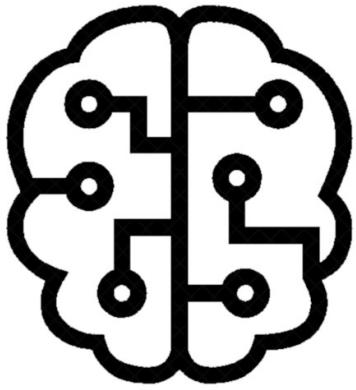
RTU и други



Philipp Slusallek RTRT on FPGA

NPU

Нервоморфни системи (NPU)



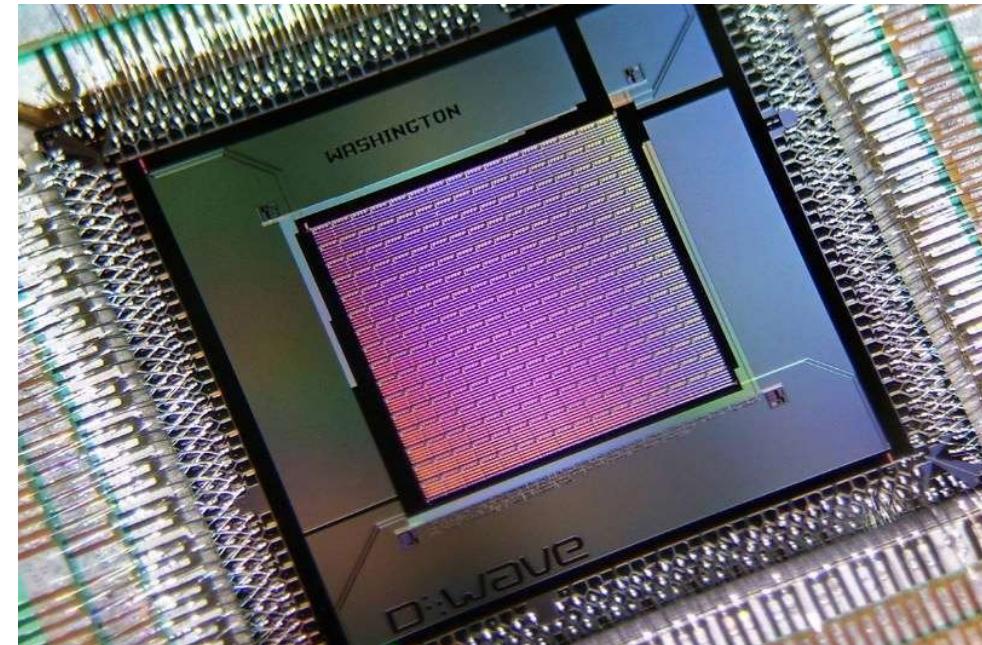
NPU

- ❖ NPU = Neural Processing Unit;
- ❖ Могат да симулират работата на много сложни невронни мрежи;
- ❖ Наричани са още AI акселератори;
- ❖ Съдържат милиарди MOSFET транзистори;
- ❖ Имат dataflow архитектура;
- ❖ Поддържат аритметика на реални числа с ниска точност;

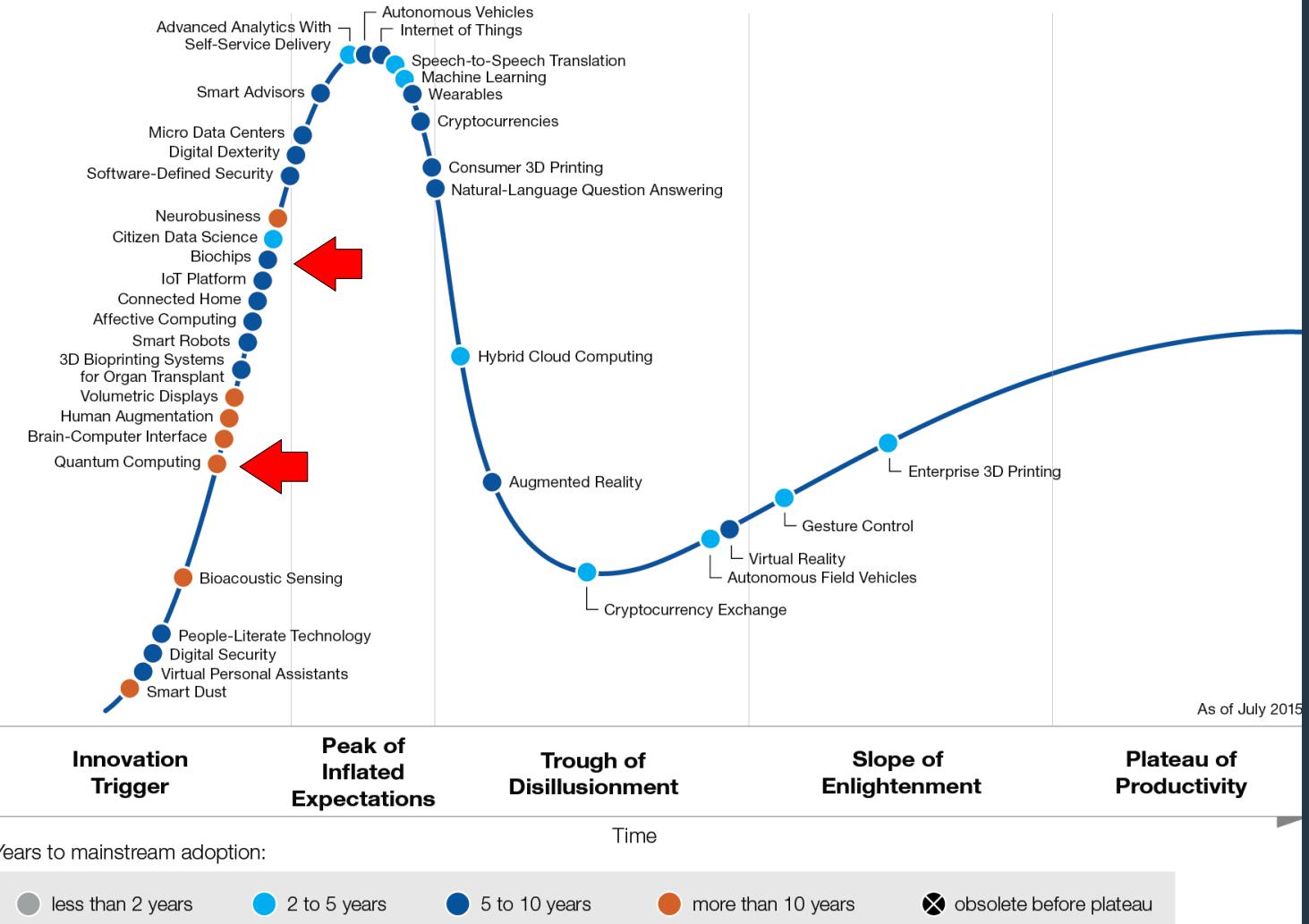
QPU.

Квантов паралелизъм

Квантови Компютри



Emerging Technology Hype Cycle



The three known types of quantum computing and their applications, generality, and computational power.

Quantum Annealer

The quantum annealer is least powerful and most restrictive form of quantum computers. It is the easiest to build, yet can only perform one specific function. The consensus of the scientific community is that a quantum annealer has no known advantages over conventional computing.



A very specialized form of quantum computing with unproven advantages over other specialized forms of conventional computing.

APPLICATION Optimization Problems

GENERALITY Restrictive

COMPUTATIONAL POWER Same as traditional computers

Analog Quantum

The analog quantum computer will be able to simulate complex quantum interactions that are impossible for any known conventional machine, or combinations of these machines. It is conjectured that the analog quantum computer will contain somewhere between 50 to 100 qubits.



The most likely form of quantum computing that will first show true quantum speedup over conventional computing. This could happen within the next five years.

DIFFICULTY LEVEL

APPLICATIONS Chemistry, Materials Science, Optimization Problems, Sampling, Quantum Dynamics

GENERALITY Partial

COMPUTATIONAL POWER High

Universal Quantum

The universal quantum computer is the most powerful, the most general, and the hardest to build, posing a number of difficult technical challenges. Current estimates indicate that this machine will comprise more than 100,000 physical qubits.



The true grand challenge in quantum computing. It offers the potential to be exponentially faster than traditional computers for a number of important applications for science and businesses.

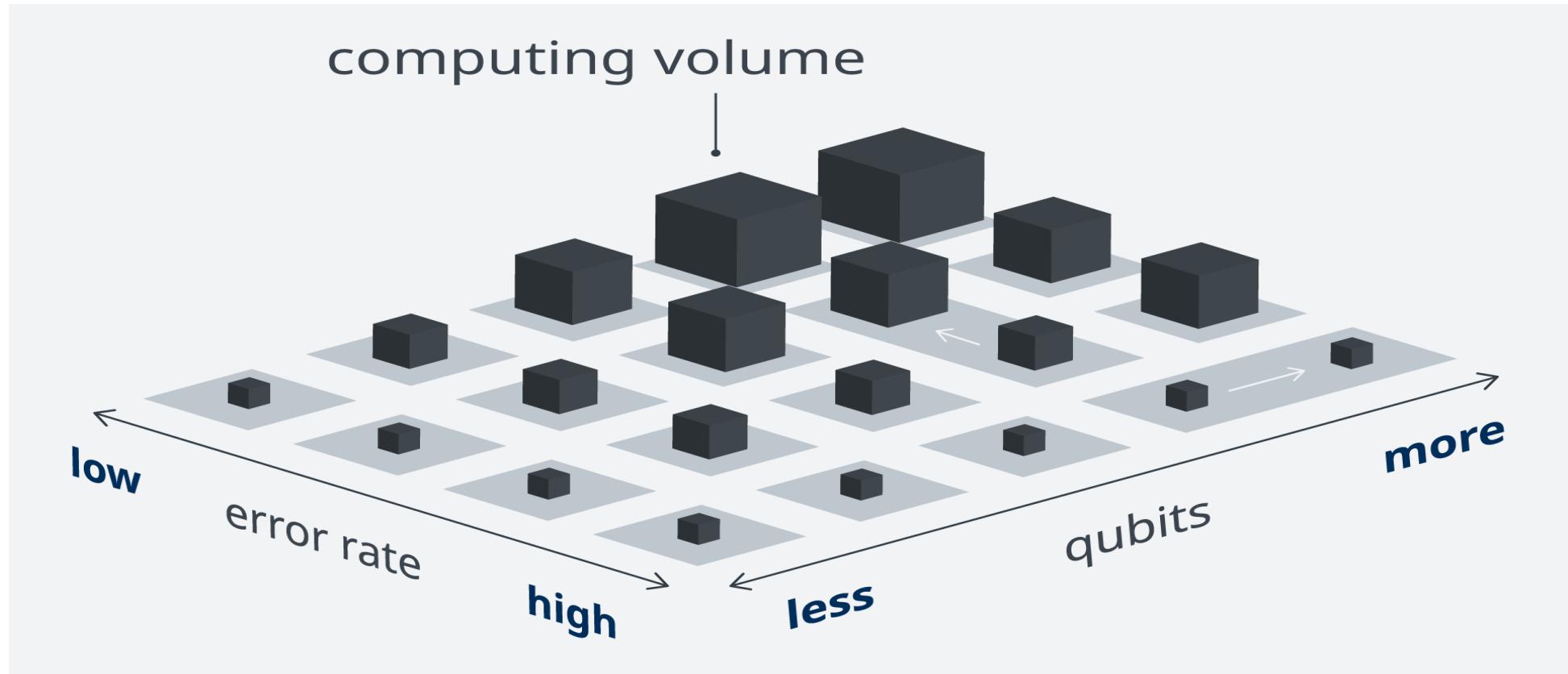
DIFFICULTY LEVEL

APPLICATIONS Secure computing, Machine Learning, Cryptography, Quantum Chemistry, Material Science, Optimization Problems, Sampling, Quantum Dynamics, Searching

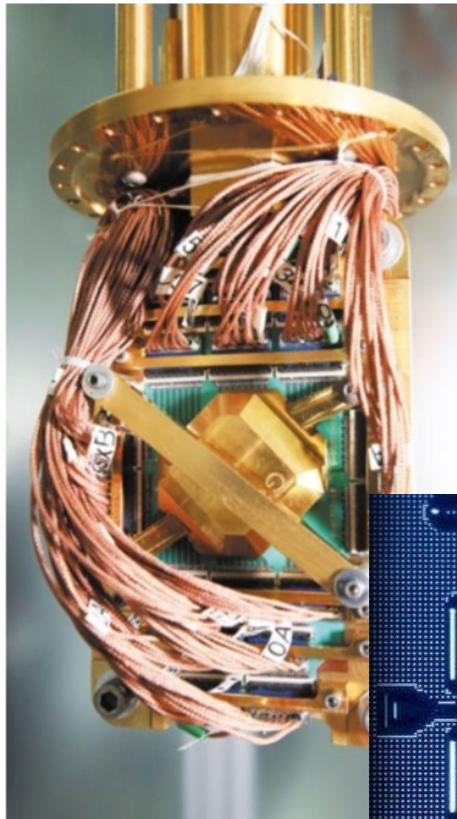
GENERALITY Complete with known speed up

COMPUTATIONAL POWER Very High

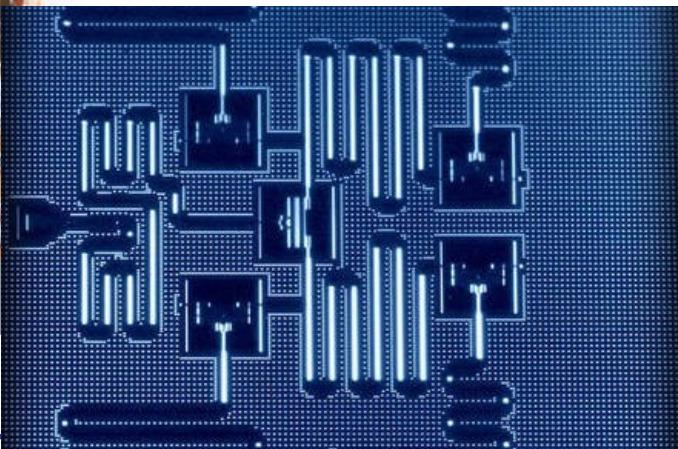
Квантови Компютри



Квантови Компютри



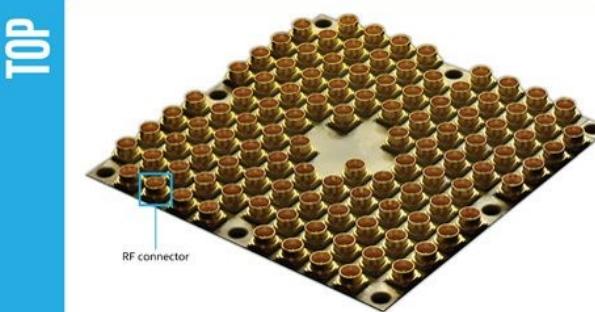
5 Qubit-ов процесор
на IBM



INTEL'S 49-QUBIT PROCESSOR

During his keynote at CES 2018 in January, Intel CEO Brian Krzanich unveiled our 49-qubit superconducting quantum test chip, code-named "Tangle Lake." The 3-inch by 3-inch chip and its package is now in the hands of Intel's quantum research partner QuTech in the Netherlands for testing at low temperatures. Quantum computing is heralded for its potential to tackle problems that today's conventional computers can't handle. Scientists and industries are looking to quantum computing to speed advancements in areas like chemistry or drug development, financial modeling, and even climate forecasting.

TOP



WORTH ITS WEIGHT IN GOLD

There are 108 radio frequency (RF) connectors on Tangle Lake that carry microwave signals into the chip to operate the quantum bits (qubits). They are made of gold, which is excellent for anti-corrosion and signal transmission.

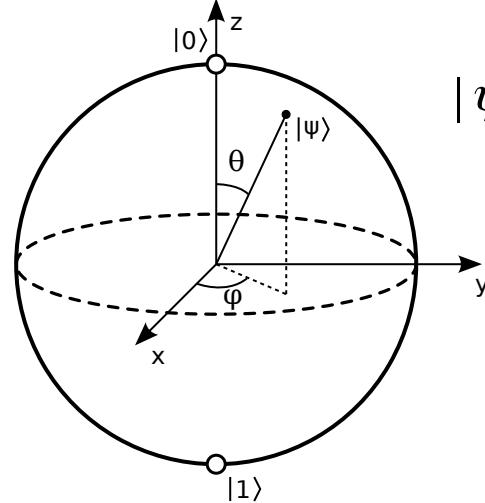
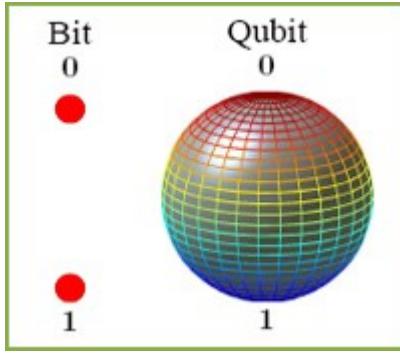
49 Qubit-ов процесор на Intel

Квантови Компютри

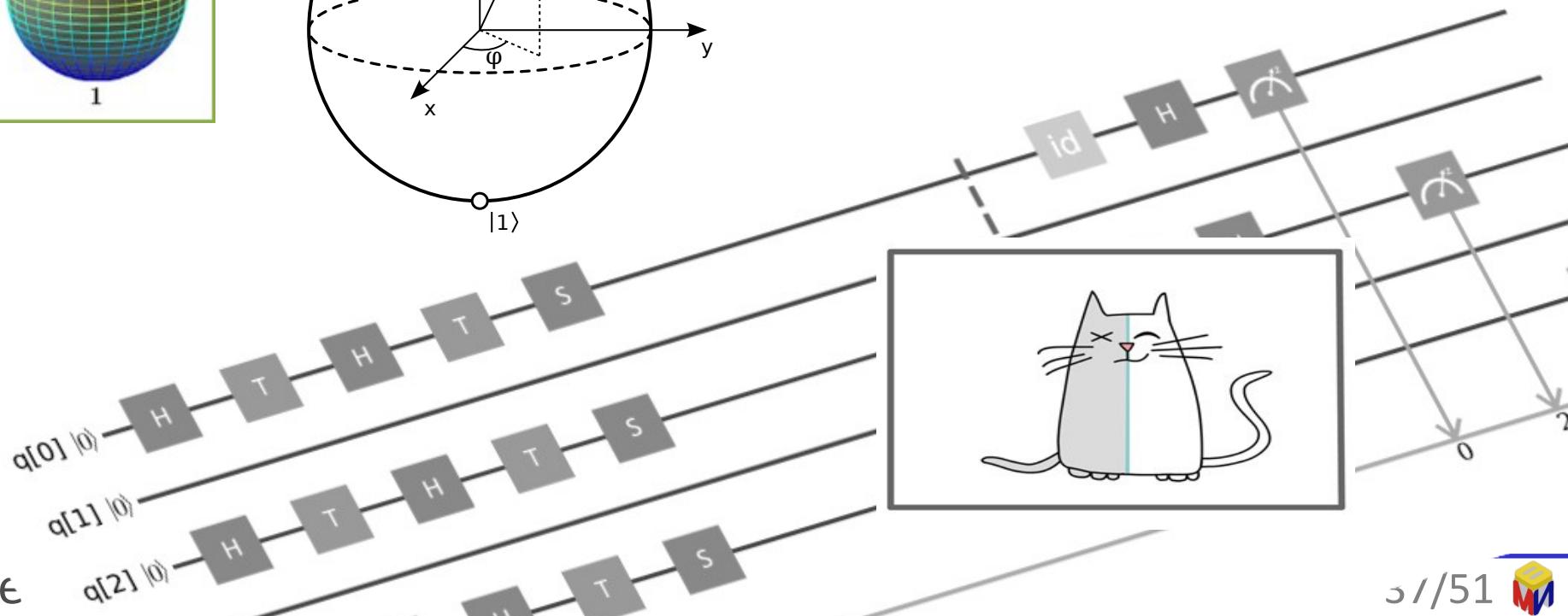


72 Qubit-ов процесор на Google

Битове, Кубити, Комката на Шрьодингер и Квантов Паралелизъм



$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, |\alpha|^2 + |\beta|^2 = 1, \alpha \in \mathbb{C}, \beta \in \mathbb{C}$$



Квантов Паралелизъм

Квантов паралелизъм (quantum parallelism) е основният принцип на работа на квантовия компютър. Според него квантовият компютър може да се представи като работещи паралелно класически компютри. В математическия модел броят на паралелните класически компютри може да се приеме за безкрайен, поради което се доказва, че квантовият паралелизъм позволява да се решат задачи, принципно нерешими с класически компютър...

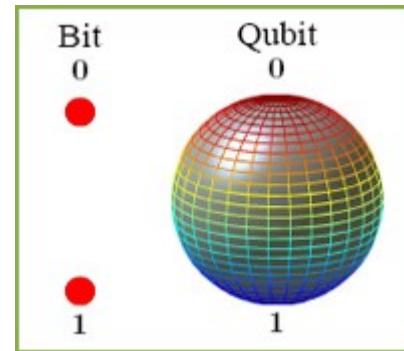
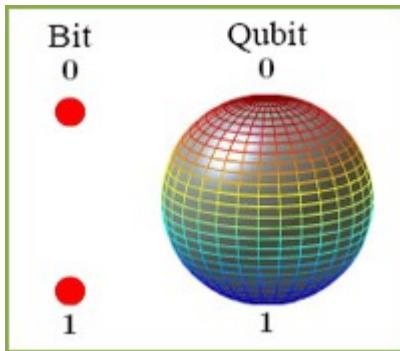
Квантов Паралелизъм

... Обаче.

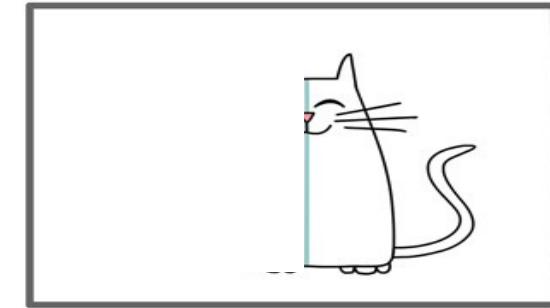
В технически осъществим квантов паралелизъм, броят на еквивалентните паралелно работещи квантови компютри, е краен. Поради това няма задача, която може се реши чрез технически осъществим квантов паралелизъм и да не може да се реши с класически компютър, макар и с цената на експоненциално забавяне в някои случаи.

Допълнително ограничение за приложението квантовия паралелизъм е извеждането на един или краен брой резултати. За сравнение, паралелно работещите класически компютри могат да изведат толкова на брой резултат, колкото е броят на паралелните клонове.

Квантов Паралелизъм



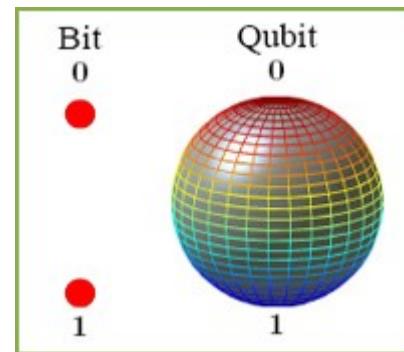
четене



За съжаление, когато прочетем даден qubit и получим нормален bit, то той е или 0 или 1 и то с някаква степен на вероятност/точност.

в 2 от 2

в 1 от 2 възможни
състояния

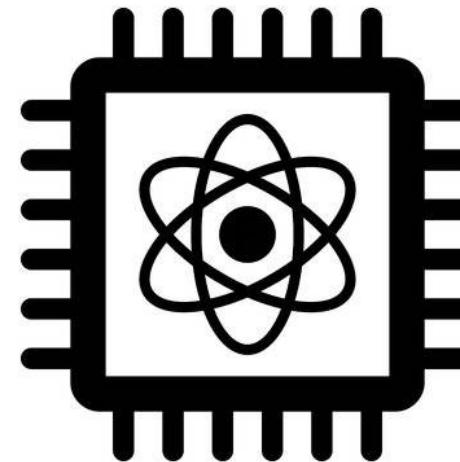
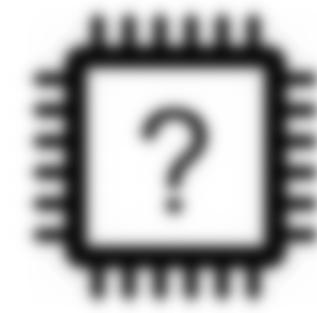
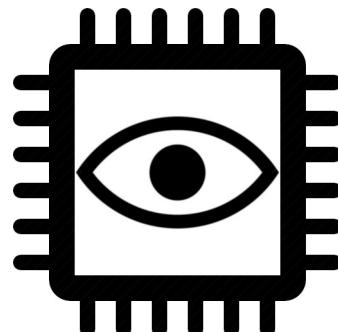
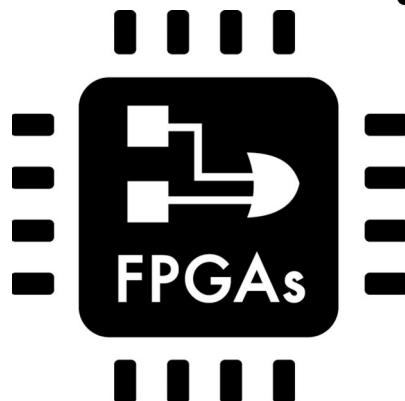
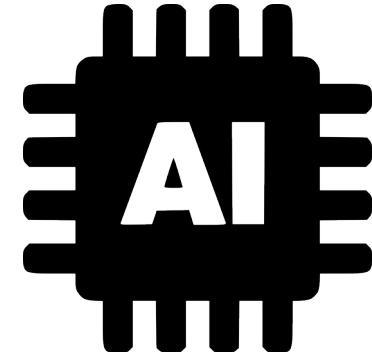
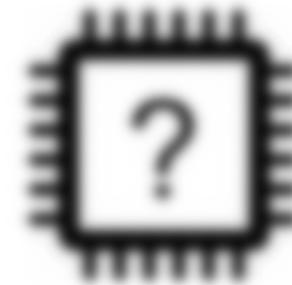
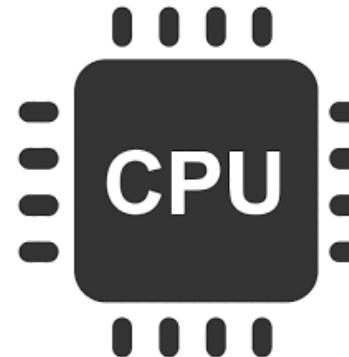
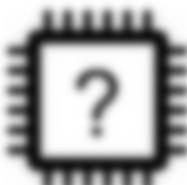
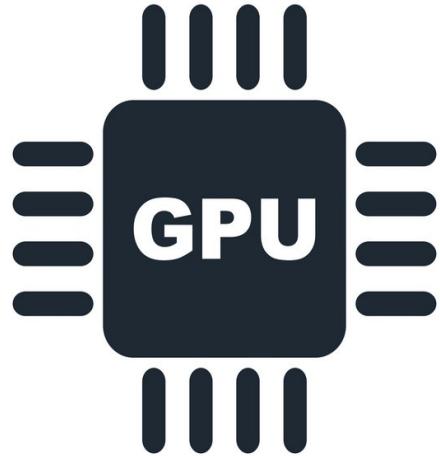


в 4 от 4

в 1 от 4 възможни

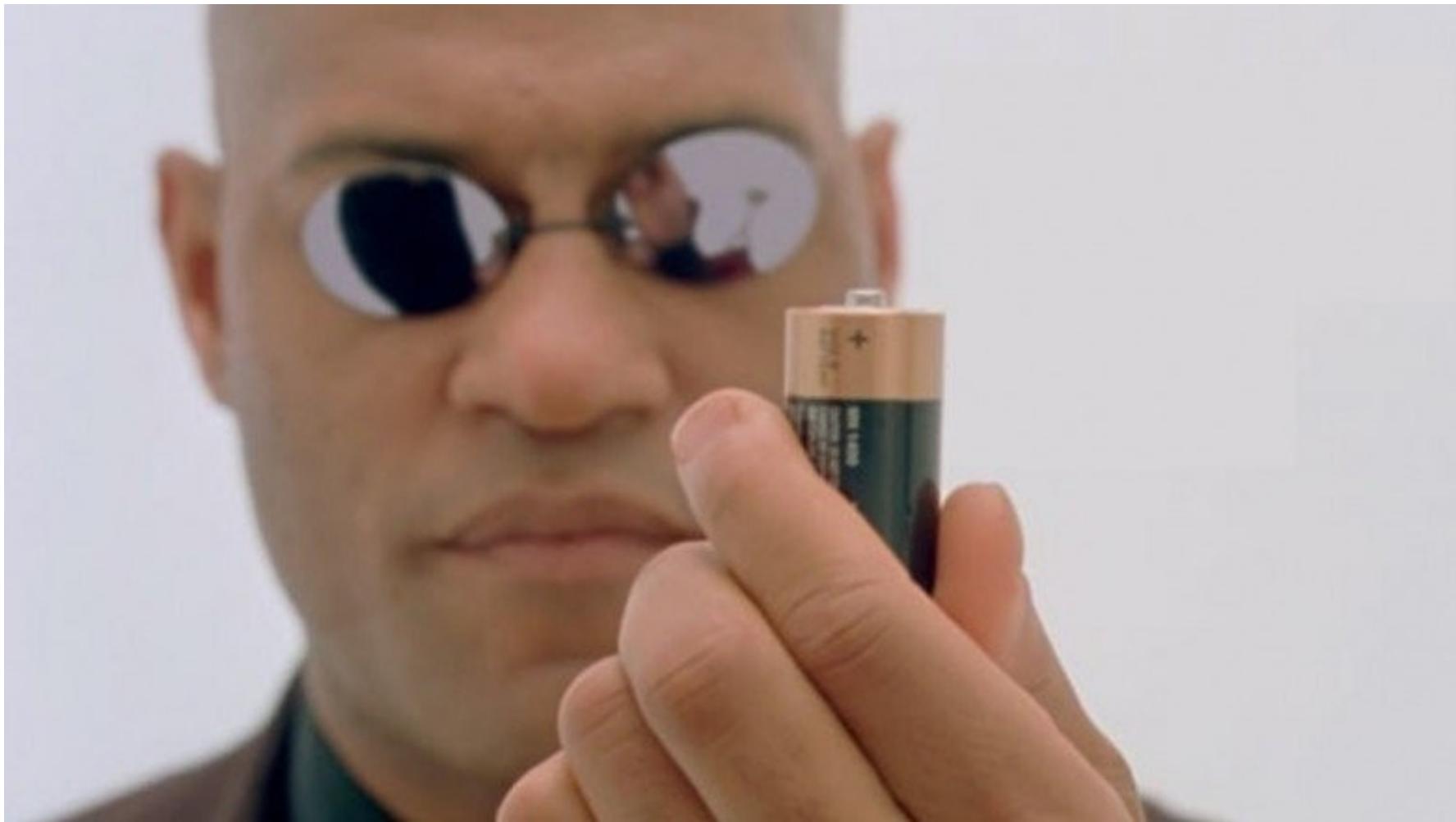
Изчислители

„Изчислители“



Изчислители

- ❖ На практика в момента се използват много разнообразни “изчислители”, базирани на най-различни технологии и архитектури;
- ❖ Всеки един от тях е добър за някои области на приложения и не толкова добър за други, където другите изчислители може да са силно приложими;
- ❖ Очертава се едно силно хетерогенно бъдеще;
- ❖ Не е въпросът дали ще възникнат и други изчислители. Въпросът е кога ще възникнат и за какви приложения/области ще са удачни;
- ❖ Не на последно място трябва да кажем, че человека (и то не от вчера) също е “изчислител” – сравнително бавен за някои задачи, но (все още) много ефективен за други. Това му отрежда място в сегашния, а може би и бъдещият хетерогенен свят от изчислители, които си взаимодействват и се допълват взаимно;



Съвременните "AI" могат да решават задачи, които преди 10 години бяха решавани само от хора, но все още хората са много по-енергоефективни за повечето от тях.

Малък експеримент – вижте този кадър...



Малък експеримент



Въпросът не е дали сте видели „Жената в червено“, а дали сте видели президентът на РБ Румен Радев?

Един новинарски „AI“ веднага щеше да напише статия „Румен Радев се разхожда сред народа“, а вие...?

Бъдеще на Паралелното Програмиране

Паралелно програмиране – развитие

- ❖ Паралелните възможности на конвенционалните ЕП;
- ❖ Домейн-специфични езици за програмиране на хетерогенните изчислители;
- ❖ Автоматизираната оптимизация, векторизация и разпаралеляване;
- ❖ Специализирани визуални и текстови ЕП за различните изчислители, например “квантови” ЕП от ниско и високо ниво и др.;

Навлизане на ПП в конвенционалните ЕП

PARALLELISM IN MAINSTREAM LANGUAGES

- ▶ Enable more programmers to write parallel software
- ▶ Give programmers the choice of language to use
- ▶ Parallel computing support in key languages



*И какво още? ...
Бъдещето ще покаже!*

Въпроси?

apenev@uni-plovdiv.bg