Паралелно Програмиране

Видове паралелизъм.

Векторизация.

Задачи – фибри, нишки, процеси...

Видове многозадачност.

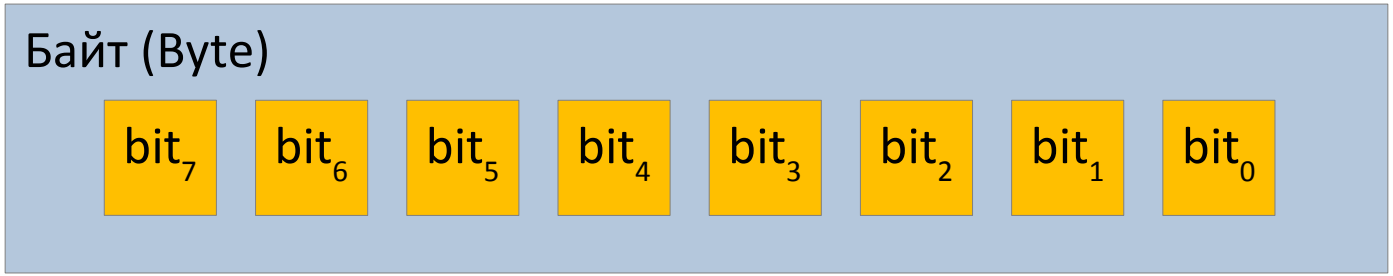
доц. д-р Александър Пенев

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);

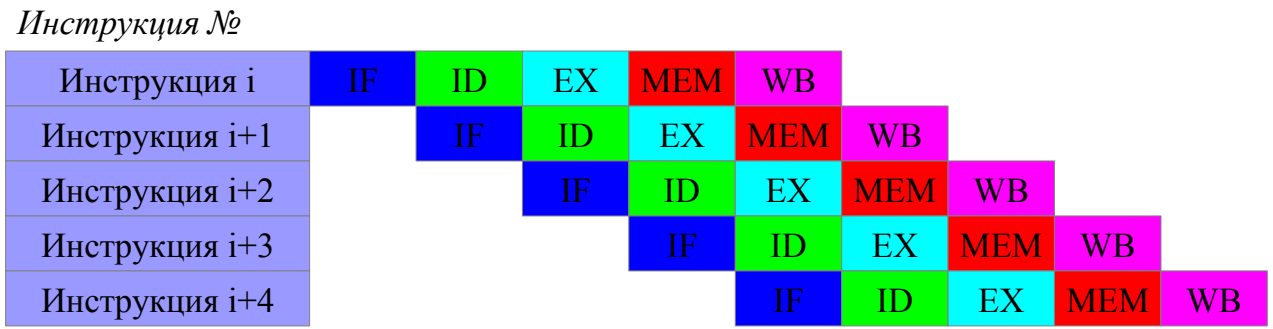
Bit-Level Паралелизъм (BLP)



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)

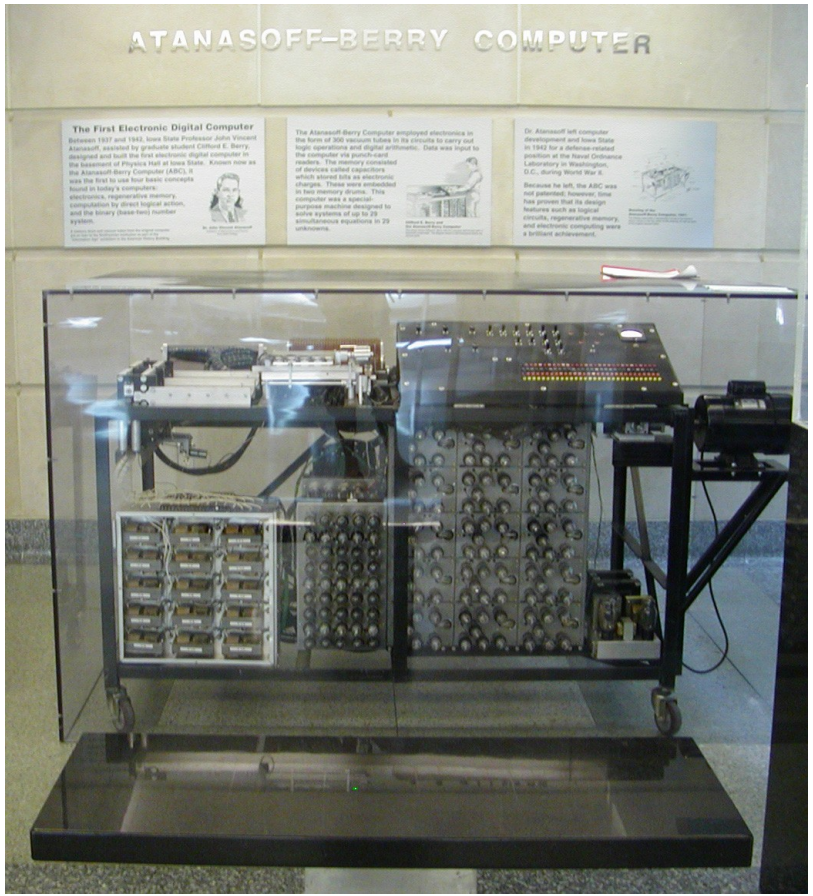


Pentium 4 с 34 стъпков pipeline

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 pedicting, ...;
- ILP не трябва да се бърка с конкурентно изпълнение (изпълнение на последователности от инструкции в отделни нишки);

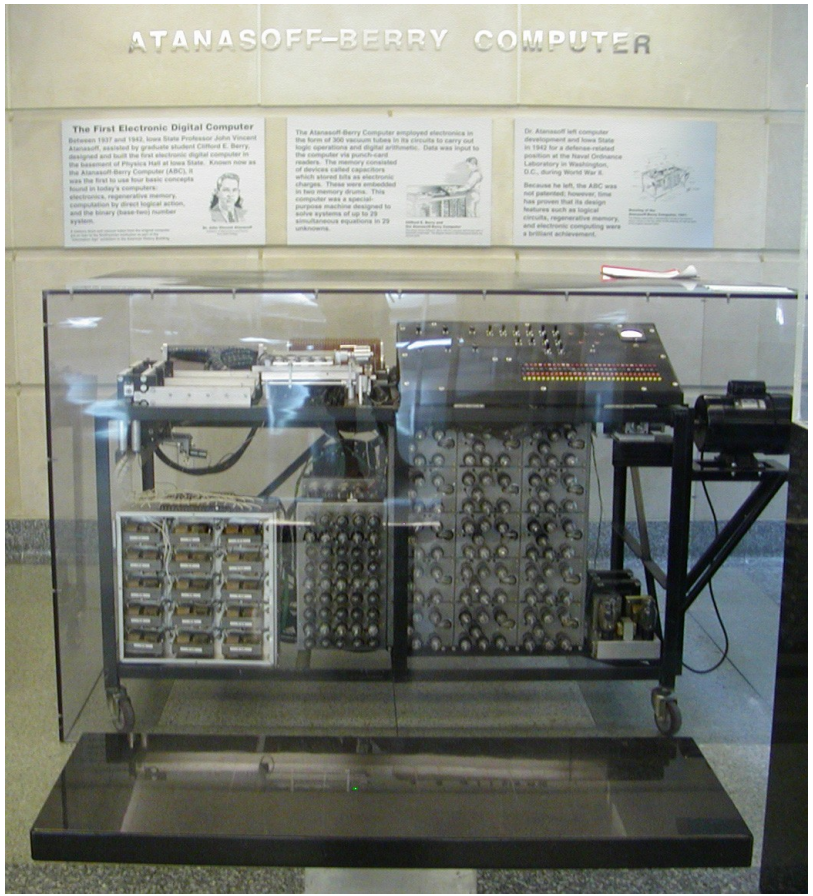
Първият компютър с паралелна обработка



Първият компютър с паралелна обработка

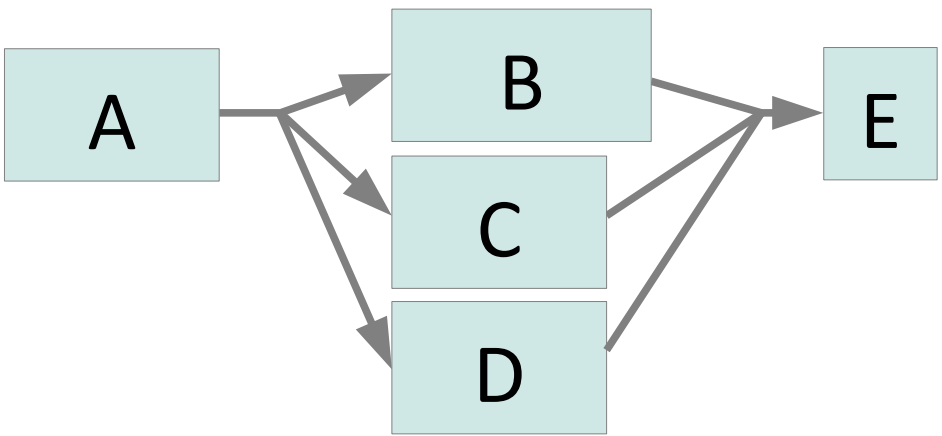
Компютърът ABC

- Не е нито програмируем, 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.

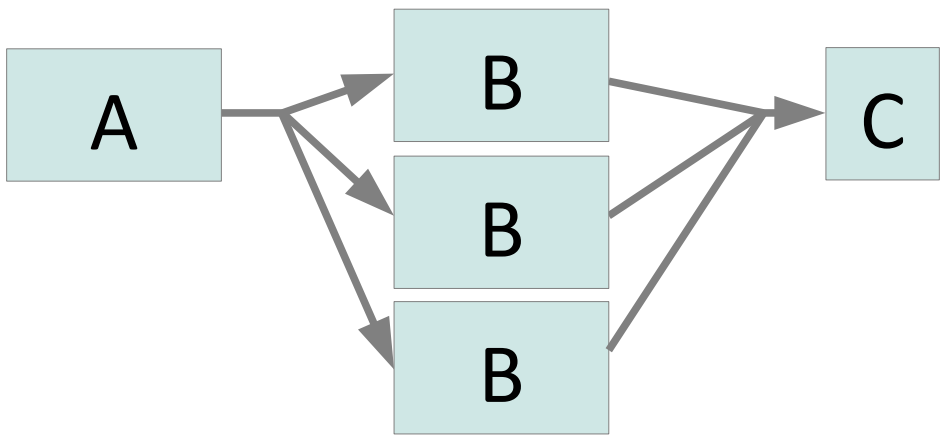


Функционален и Даннов Паралелизъм

- 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

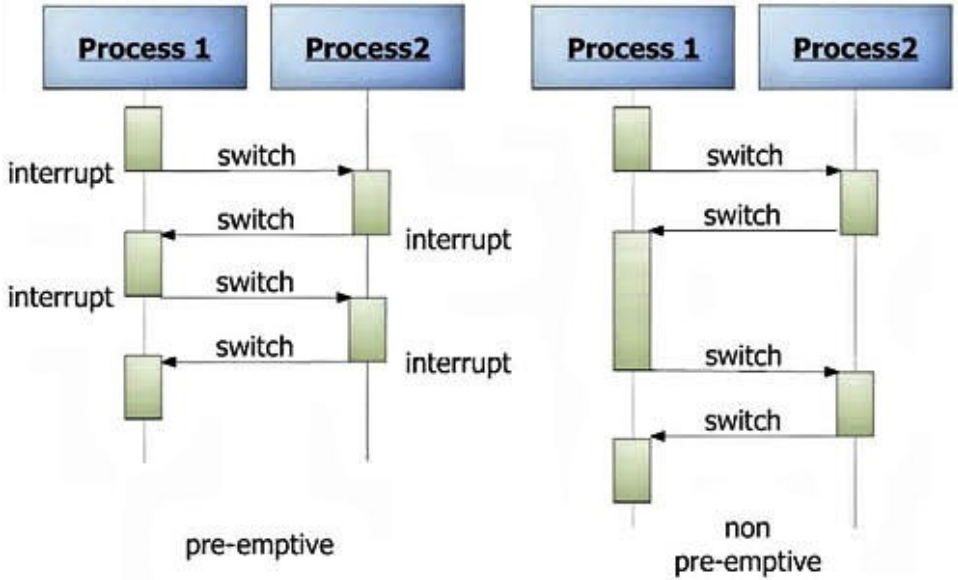
Задачи

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

Видове:

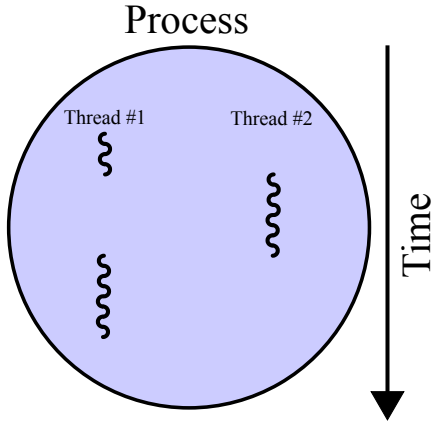
Фибри (fibers, coroutines), Нишки (Threads), Процеси (Processes);

Фибри



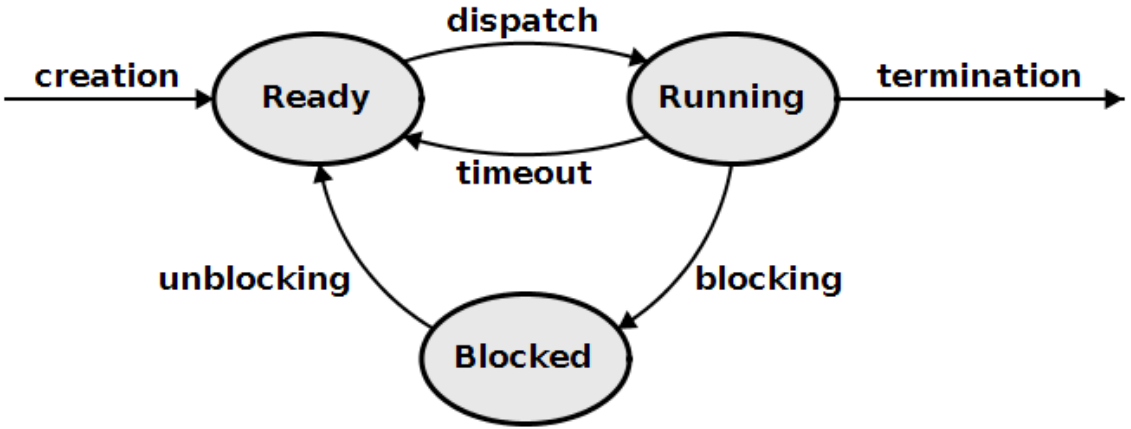
- Фибри (Fibers) – Леки нишки; Споделят адресното си пространство; За разлика от нишките фибрите използват кооперативна многозадачност, а не „превантивна“ многозадачност (preemptive multitasking);
  Coroutine е синоним;
  Могат да има състояние или да нямат;

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.;

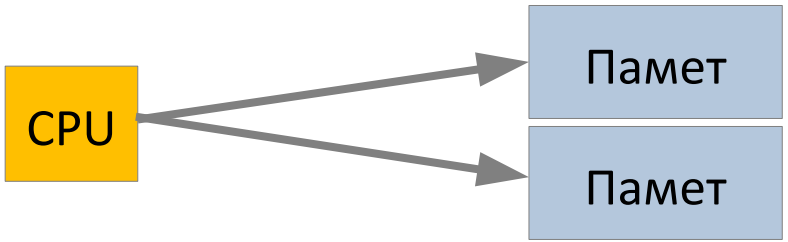
Задачи

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 компютърните 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

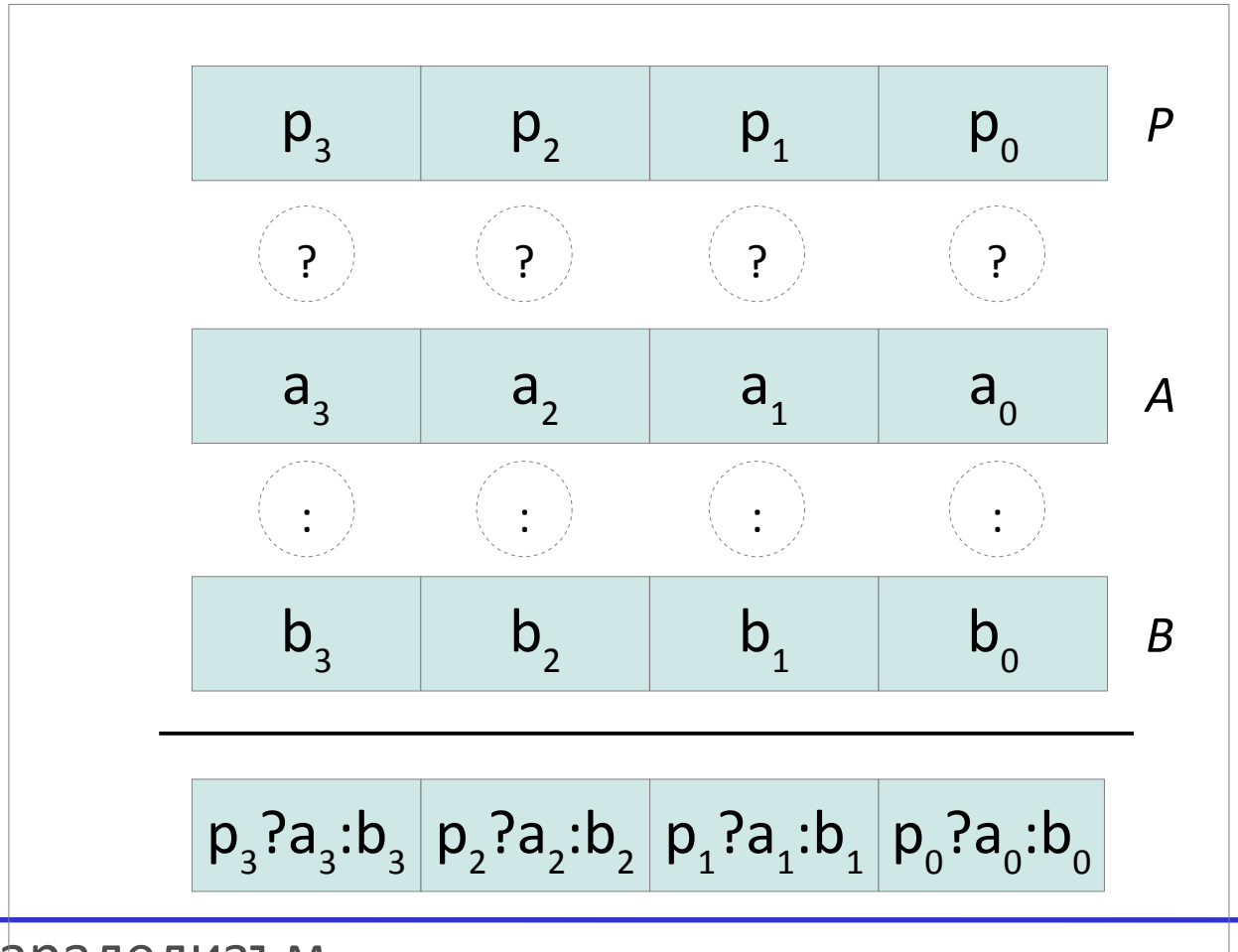
```
for (i = 0; i < 1024; i++)
{
  if (A[i] > 0)
    C[i] = B[i];
  else
    D[i] = D[i-1];
}
```

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 ф-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(и) на цикъла (циклите)

- След цикъл

  Получаване на резултати и допълнителна инварианта 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];
}
```

ОК.
Цикълът може да бъде развит и/или векторизиран напълно

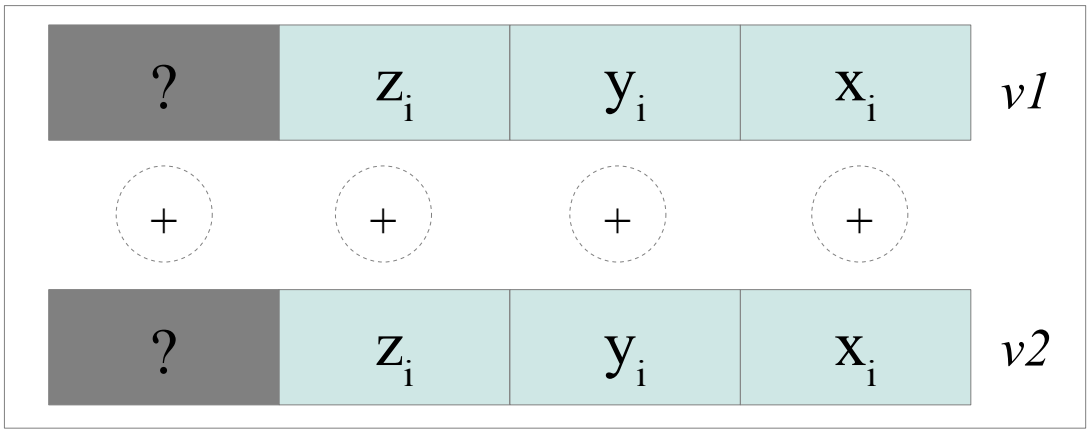
```
for (i = 0; i < 1024; i++)
{
  C[i] = A[i] + B[i];
}
```

ОК.
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?

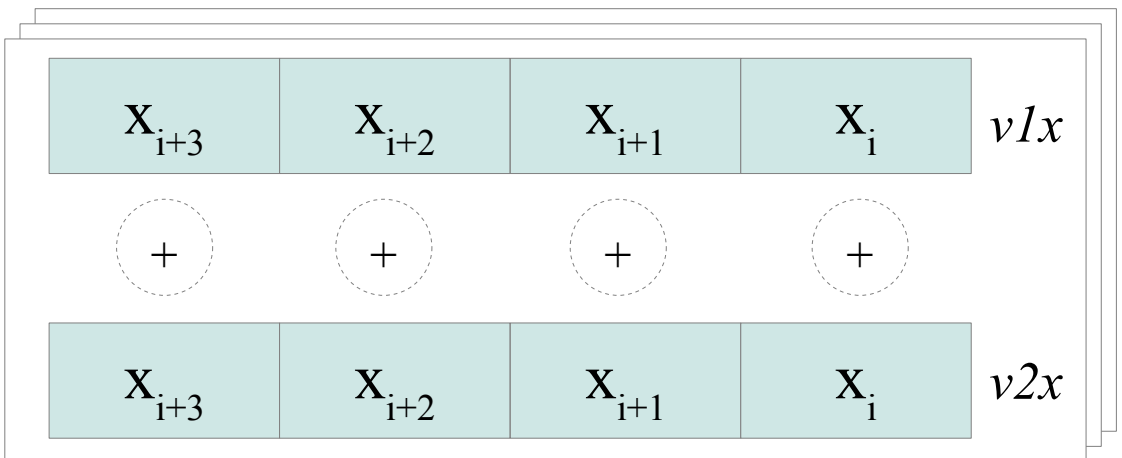



```
struct { float x, y, z; } vec;

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



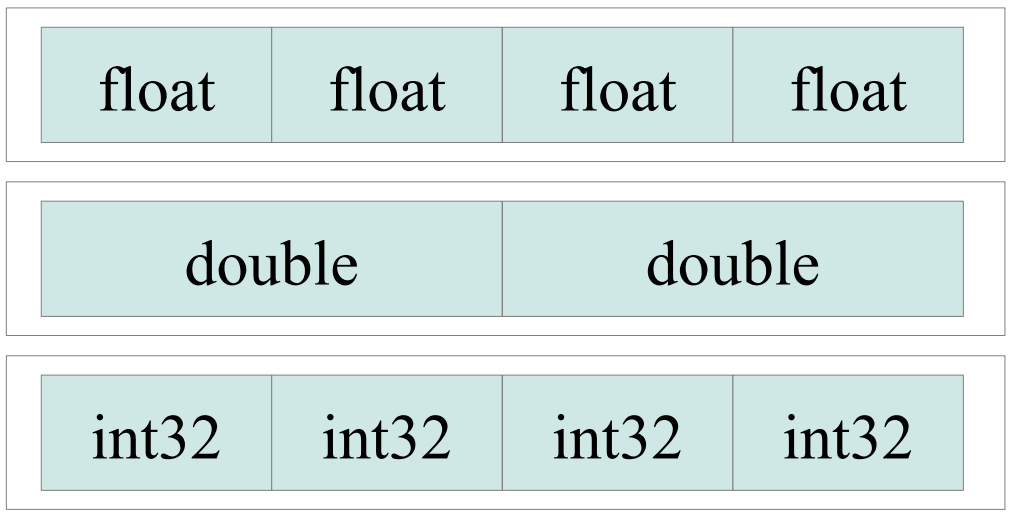

```
float[100] v1x, v2x;
float[100] v1y, v2y;
float[100] v1z, v2z;

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

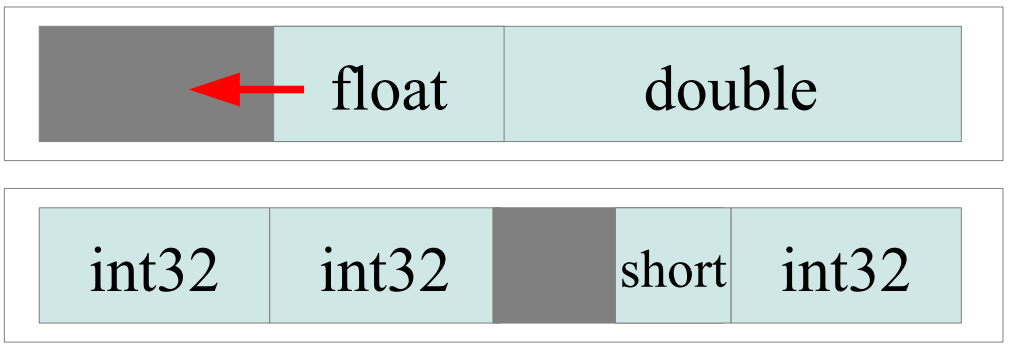
  Например 4 float числа се събират точно в един SSE регистър



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.

Многозадачност

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 TMT:
  - 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 многозад. (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;

Клъстерирана многозад. (Clustered multithr. – CMT)

При **Клъстерирана 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 оползотвори неизползвания хардуер при изпълнение на повече задачи едновременно;

Спекулативна многозад. (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 многозад. (Cooperative multithr.)

Кооперативната многозадачност (също извеснтна 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