

1. Модели компютърни архитектури

- Модели машинна архитектура и обработка: класификация и метрика
- Мултипроцесори: **UMA, NUMA, SOMA**
- Векторни и потокови машини и систолични матрици
- Мултикомпютри

Класове компютърни архитектури

1.3

- архитектура – компоненти и организация на системата
- фон Нойманова (1.3.1) – на възли и мрежи и неklasическа организация (систолични, потокови, логически и редуционни модели и невронни мрежи)
- класификация на **Michael Flynn (1966)** по управление на потока инструкции и потока данни (операнди) – **SISD, SIMD, MISD и MIMD** архитектури – (1.3.2)
 - **SIMD** – за векторна обработка, фина грануларност
 - **MISD** – за конвейерна обработка (обработващи фази върху вектор) – систолични масиви
 - **MIMD** – обикновено с локална и глобална памет; за средна и едра грануларност
- технологично-ориентирана таксономия на паралелните архитектури: мултипроцесори, мултикомпютри, потокови машини, матрични процесори, конвейерни векторни процесори и систолични матрици – частично съответствие с класовете на Флин (1.3.3)

HW/SW паралелизъм

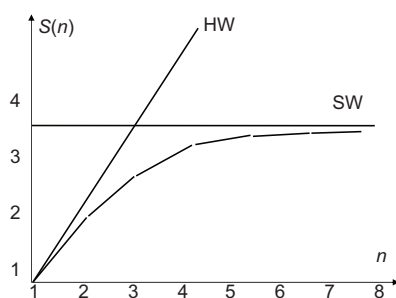
1.4

- За паралелно изпълнение на програми е необходима едновременно апаратна и програмна поддръжка.
 - Апаратен паралелизъм. Обуславя се от архитектурата и ресурсите, които са баланс между производителността и цената. Характеризират се с пикова производителност и средно натоварване. Той задава зависимостта по ресурси.
 - Програмнен паралелизъм Обуславя се от зависимостта по данни и по управление. Реализира се като
 - паралелизъм по управление - конвейеризация, мултиплициране на функционални възли. Обслужва се паралелно, прозрачно за програмиста.
 - паралелизъм по данни - типичен за SIMD, но и при MIMD.

Метрика: ускорение и ефективност

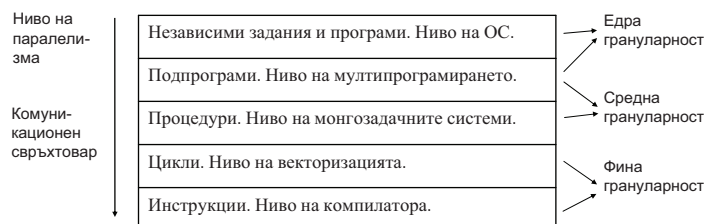
1.5

- ускорение $S(n) = T1/Tn$; лимитиращи фактори
- ефективност $E(n) = S(n)/n$



Делене на обработката: грануларност

1.6



SIMD

1.7

- обобщения модел включва контролно устройство и еднотипни обработващи модули с достъп към обща памет – (1.7.1)
- програмно-апаратна зависимост на паралелизма/ускорението – пример за изпълнение на програма на SIMD машина (1.7.2)
- процесорните елементи изпълняват операциите във формат битове или думи
- локалната памет за данните може да бъде разпределена, обща или йерархична (със свързваща мрежа) (1.7.3)
- особености:
 - опростена архитектура спрямо MIMD поради общото контролно устройство (за дешифриране и зареждане на инструкциите) и съответно поддръжане само на едно копие от кода за инструкции
 - скаларните операции (включително контролната логика) се изпълняват от контролното устройство – евентуално конкурентно на паралелната обработка на данни в обработващите устройства
 - имплицитна синхронизация между отделните обработващи устройства (при MIMD – експлицитна)
- примери – фамилия **Connection Machine** на **Thinking Machine Co.**

MISD

1.8

- това е архитектурния принцип на всички конвейери – вкл. на процесорния конвейер – обработката се разделя на последователни фази; обработката на следващата инструкция (при най-фина грануларност) или на следващия процес започва веднага щом предходния процес освободи първата фаза – (1.8.1)
- прилагат се и функционални (или циклични) конвейери например с фазите (1.8.2):
 - четене на инструкциите от обща памет
 - зареждане в обработващото устройство с евентуално буферизиране
 - обработка
 - пренос на резултата към общата памет (буферизиране)
 - запис в общата памет
- инструкционно, субсистемно (обикн при аритметична обработка – нелинейни конвейери с фази **add, mul, div, sort...**) и с т е м н о н (процеси, също и програмна организация) на конвейеризация

- представяват модификация на MISD на субсистемно ниво, специализирана архитектура за определени алгоритми – с многодимензионни конвейри т.е. фиксирана мрежа от обработващи устройства
- ограничено приложение – ЦОС (цифрова обработка на сигнали – DSP), обработка на образи и др.
- опростени процесорни елементи и комутираща съобщителна мрежа с ограничен набор шаблони
- управлението е по инструкции (control flow – не data flow) но програмирането е като при потоковите архитектури
- архитектурата включва обработващ масив (с комутатор) и управляващ модул, който настройва масива, предава данните и извлича резултатите (+ контролен възел – хост) – (1.9.1)
- производителността се понижава значително при интензивен вход/изход
- топологични шаблони:
 - систолични вектори – по същество конвейри
 - двудимензионни масиви – обикновено регулярни с коеф. на съседство най-често 4 или 6 (1.9.2)

MIMD

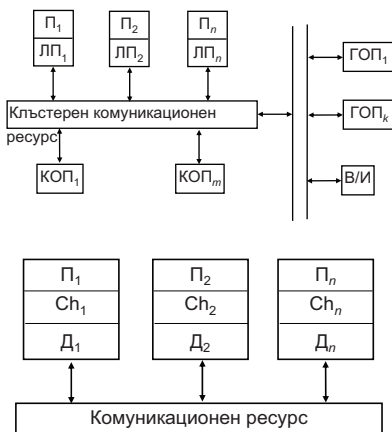
- това е архитектурния принцип на всички мултипроцесори и мултикомпютри:
 - процесорите са автономни и могат да изпълняват различни програми (вкл. локално копие на ОС)
 - имат общ ресурс с разпределен конкурентен достъп – памет или комуникационна среда
 - организация: по памет / по комуникации
- | | |
|---------------------------|--|
| автономни (локална памет) | общо адресно пространство (общодостъпна памет) |
| магистрални | комутиционни |
- универсални, отказоустойчиви, по-едра грануларност
 - обикновено се изграждат с масови процесори (вместо специализирани процесорни елементи с ограничени функции)

Мултикомпютри

- Разпределената обща памет (distributed shared memory DSM): програмната имплементация на обща памет в система с автономни възли (и адресно пространство)
- виртуално общо адресно пространство от страници (не думи) – 4/8 kB – (което позволява програмиране за мултикомпютъра като за виртуален уникмпютър)
 - при отсъствие на страница от локалната памет възниква вътрешно прекъсване (memory trap) и зареждане на страницата в локалната от отдалечената памет
 - възможно е репликиране на страници само за четене (read only);
 - ако страницата е и за запис, се прилагат различни мерки за поддържане на свързаност
- Системи с обмен на съобщения – Message passing distributed systems

NUMA и COMA

- NUMA (non-uniformly shared memory access) – йерархия на общата памет – локални, глобални и/или клъстерни памет
- COMA (cache only shared memory access) – паметта е локална (cache) но йерархията и позволява част от нея (“директория”) да се адресира отдалечено.



...Систолични матрици (Systolic Arrays)

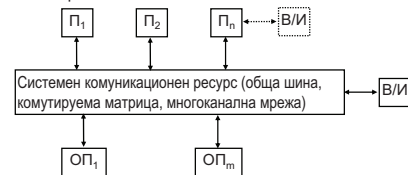
- тенденцията е към елементи за фина грануларност – на инструкционно ниво – снабдени с няколко високоскоростни дуплексни серийни канали (броя на които определя валентността – коеф. на съседство)
- пример: iWrap серия на Интел и университета Carnegie-Mellon (1.10.1) – процесорната клетка се състои от
 - iWrap компонент с изчислителен и комуникационен агент и
 - страницирена памет с директен интерфейс към компонента
- пример: умножение на матрици в двумерен систоличен масив с коеф. на съседство 6 (1.10.2)

...MIMD

- наличието на автономна локална памет ги разделя на:
 - системи с обща памет; синоними: мултипроцесори | [shared-memory | tightly-coupled] systems | Global-Memory MIMD, GM-MIMD | Uniform Memory Access System – UMA
 - системи с обмен на съобщения; синоними: мултикомпютри, [distributed-memory | loosely-coupled] systems | Local-Memory MIMD, LM-MIMD | Non-Uniform Memory Access System – NUMA (поради наличието на локална и отдалечена памет)
- глобално и локално адресно пространство; виртуалната памет поддържа глобално адресно пространство на страниците (не на ниво думи), което се управлява от разпределена ОС (РОС) за МП и хомогенните МК. При МК общата виртуална памет се поддържа и с обмен на съобщения – 1.12.3
- хетерогенните МК използват мрежови ОС (МОС), при които нивото на достъп е разпределена файлова система (напр. базирана на DNS) с ползване на примитиви от типа rlogin, rcp...

Архитектура с обща памет (мултипроцесори) – UMA

- UMA (uniformly shared memory access) – еднакъв достъп на процесорите – силно свързани системи: архитектура
 - обща шина – разширение от унипроцесинг към мултипроцесинг
 - комутируема матрица (crossbar switch)
 - многоканални мрежи



- синоним: симетричен (централизиран В/И) и асиметричен (специализиран процесор за В/И) мултипроцесинг – обикновено хомогенни системи.

Потокови архитектури (Data Flow)

- при класическите фон Нойманови архитектури (вкл. модификациите по Флин) програмата е последователност от инструкции, която се изпълнява от контролно устройство – control flow
- при потоковите архитектури операциите се изпълняват веднага при наличие на операндите (и наличие на операционен ресурс) – контрола се осъществява чрез планиране на операндите т.е. данните; концептуално всички инструкции с готови операнди могат да се изпълняват паралелно (на практика конкурентно)
- програмите за потокови архитектури се представят с потокови графи (обикн. с текстов синтаксис) – възлите представят операции, а дъгите – информационните връзки на операндите; нивото на паралелизъм обикновено е инструкционно – 1.16.1
- допълнителни особености на потоковите архитектури: реконфигурация, буферирание на данните, комплементиране на операндите
- наличие на управляващ процесор, който пакетира операндите и инструкцията в блок – token – и го предава на някой от обработващите процесори

- **статични** – програмния (потоковия) граф е фиксиран. За изпълнение на повече от една програма се използват различни варианти на зареждането на данните, които се генерират на етапа компилация
 - този модел не поддържа процедури, рекурсия и обработка на масиви
 - организация 1.17.1
- **статични с реконфигурация** – логическите връзки между процесорните елементи се установяват на етапа зареждане на програмата: топологията на връзките се решава от компилатора и след зареждане на програма остава фиксирана при изпълнението; особености:
 - физическите канали съществуват, но са комутират
 - броя алоцирани (заредени) процесори обикновено е по-малък от инсталираните процесори поради ограничения в комутицията – логическата връзка между процесорите е дърво, не всички процесори в листата на което се използват
 - пример – MIT Data Flow Machine – клетките памет съответстват на информацията във възлите на потоковия граф – т.е. инструкционните блокове (tokens) – когато блока е комплектован с операнди, той се предава като операционен пакет към елемент за обработка; пакета с резултата се връща в клетъчната памет – 1.17.2

- базират се на логически канали между процесорите, които могат да се реконфигурират по време на изпълнение подобно на система с обмен на съобщения – с маркирани блокове (tagged tokens)
 - дъгите в потоковия граф могат да съдържат повече от един блок едновременно (но с различни марки!)
 - операциите се извършват когато възела получи блокове (с еднакви марки) на всичките си - входящи дъги
 - циклични итерации могат да бъдат изпълнявани паралелно: за целта всяка итерация се представя като отделен субграф като маркировката се разширява с номера на итерацията – 1.18.1 (само при информационна независимост на итерациите!)
 - пример – Manchester Data Flow Machine MDM: цикличен конвейер, в който блоковете циркулират и се управляват от ключов модул. Компонентите са (1.18.2)
 - Блоков буфер (token queue) – за съхраняване на междинни резултати (ако се произвеждат по-бързо отколкото е последващата им обработка) – капацитет 32К блока и производителност 2.5 Мблока/сек
 - Комплементираща памет (matching store) – за комплементиране на блоковете с еднакви марки – процеса е апаратен и поддържа до 1.25 Мблока
 - Памет инструкции (instruction store) – n-торките (обикновено 2ki) операнди-блокове се пакетират с инструкции и адрес (етикет) на резултата и се предават за изпълнение
- обработващ модул (processing unit) – 20 процесора (24-битова дума и 4Кдуми вътрешна памет)

Съпоставка на компютърните архитектури

Тип	Принцип на действие	Интерфейс	Приложимост	Сложност	Ефективност
SIMD	спонтанен	директен	средна	висока	висока
MIMD	сложна абстракция	най-сложна организация	висока (универсални)	висока	средна
MISD	спонтанен	директен	ниска	ниска	висока
Систолчни	сложна абстракция	директен	ниска	средна	висока
Потокови	сложна абстракция	сложна организация	висока	висока	висока

Характеристики на мрежите за връзка

- разстояние d_{ij}
- диаметър на мрежата $D = \max\{d_{ij}, \forall(i, j)\}$ - изисква по-голям брой канали между възлите, респ. валентност
- валентност на възлите (degree)
- сечение (bisection width) $S = \min\{\text{AllLinks}(X, Y) : ||X| - |Y|| \leq 1\}$
- разширяемост

топология	брой възли	валентност
линия и пръстен	d	2
двоично дърво	2^d-1	3
shuffle exchange	2^d	3
двудименционна мрежа	d^2	4
хиперкуб	2^d	d
пълнен граф	N	N-1

Съдържание

- Процесорни архитектури – технологично пространство
- Линейни и нелинейни конвейери
- Архитектура на набора инструкции
- Субинструкционен паралелизъм
- Суперскаларни и мултипроцесорни масови процесори

Мрежи за връзка

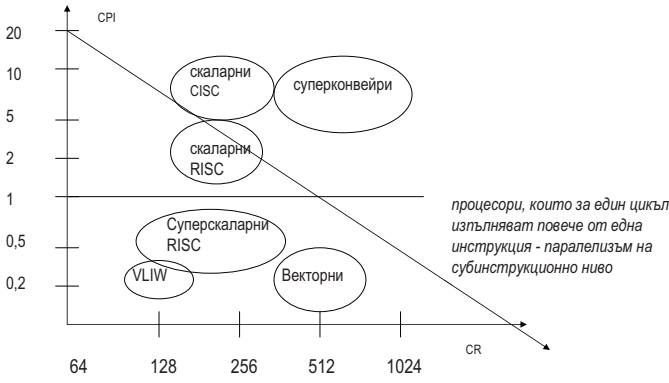
- осъществяват комуникациите между процесорните възли привлички видове МП и МК – статични и динамични (базират се на [каскади от] комутируми блокове - ключове)
- топологии на свързване
 - пълен граф
 - линия и пръстен
 - двудименционна циклична и ациклична мрежа
 - хиперкуб (n-куб)
 - двоично дърво
 - shuffle exchange

2. Процесорни архитектури

Процесорни архитектури

- Main frame – широк архитектурен клас от компютри, прилагат се някои от следните процесорни архитектури:
 - Скаларни процесори CISC (Complex Instruction Set Computer); RISC (Reduced Instruction Set Computer);
 - Суперскаларни CISC; RISC, само че RISC се използва по-често по технологични причини;
 - Процесори VLW – Very Long Instruction Word;
 - Векторни;
 - Суперконвейрни [super pipeline];
- Основни характеристики на всички архитектурни класове процесори:
 - Процесорни цикли CPI;
 - Такова честота CR.
- Тези два параметъра на пръв поглед са независими, но между тях съществува корелация, която може да се представи в диаграма на технологичното пространство:

Диаграма на технологичното пространство [MHz]



Фази на инструкционен конвейер

- Процесорната обработка на типична инструкция реализира MISD паралелизъм на инструкционно ниво и минава през фазите извличане (от cache - обикновено 1 инстр. за цикъл), декодиране (установява функцията за изпълнение и необходимите ресурси - регистри, магистрали, устройства), издаване (резервира ресурсите чрез блокиране и извлича операндите от регистрите към устройствата), изпълнение (1 или повече фази), записване (writeback - на резултатите в регистрите).

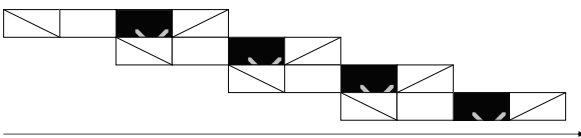


Времедиаграма на инструкционен конвейер

- Закъснението между две последователни инструкции е една фаза при скалярните процесори



- Поради ресурсен конфликт между фазите на извличане и запис по-често се прилага закъснение на две фази (два субтакта) между инструкциите:



Синхронни линейни конвейерни процесори ЛКП

- ЛКП е каскада от k процесорни фази (stages - S_i), която изпълнява фиксирана функция върху данните, преминаващи през устройството от входа (S_i) през последователните фази ($S_i \rightarrow S_{i+1}$) към изхода му S_k . Те не са динамично (runtime) настройваеми т.е. са статични. Изпълняват операционни, аритметични и обменни инструкции.
- Синхронните ЛКП са с интерфейс между фазите, който представлява синхронизиращи буферни ключови (latches) с общ такт. Фиг. 2.7. Ключовете са регистри които изолират входовете от изходите и предават данните синхронно във всички фази. Фазата с най-голямо закъснение определя общия такт и общата производителност: $\tau = \tau_{max} + d_{latch}$; $P_{peak} = f = 1/\tau$. Проявява се и фазово отместване (skew[ing]) на такта при предаване на тактовия сигнал между фазите. Затова се избира $\tau = \tau_{max} + d_{latch} + s$.

Асинхронни линейни конвейерни процесори ЛКП

- Асинхронните ЛКП контролират потока данни с "Hand Shaking" протокол - Ready/Ack между $S_i \rightarrow S_{i+1}$. Фиг. 2.8. Подходящи за комуникационни канали при системи с обмен на съобщения. Производителността на отделните фази може да варира.

Нелинейни конвейерни процесори НЛКП

- Динамични, настройваеми, допуска се разклонение, обратна връзка (feedback) и Φ предаване (feedforward) на данните за обработка. Фиг. 2.9.1. Изходът може да не е от последната фаза.
- Каранаре з е р а в ъ н и я т не е тривиална като при ЛКП. За различните функции може да варира по устройства и време (тактове). Фиг. 2.9.2. Тя се дава и съвместимостта на последователните функции по устройства т.е. зависимостта им по ресурси

Анализ на закъснението при НЛКП

- Закъснението (latency) се представя от броя процесорни тактове k между две последователни инициирания на функции.
- Опитва да повече от едно инициране едновременно на едно устройство е колизия, която се избягва чрез планиране (диспечеризация, scheduling) на последователността от инициирания.
- Когато закъснението е такова, че предизвиква колизия, то е забранено закъснение; трябва да се избере последователност от закъснения, така че да не предизвиква колизия. Пример за две забранени закъснения с карта на резервацията - фиг. 2.10.
- Цикъл на закъснението е последователност от закъснения, която се повтаря неопределено дълго. Интервалите между две последователни инициирания на функции в цикъла на закъснението може да са еднакви, (константен цикъл), но може и да са различни, при което се изчислява средно закъснение. Чрез коефициента на запълване на цикъла се получава ефективността на конвейера.

Инструкционен конвейер

- ИК е специализиран за обработка на последователните инструкции в машинния код чрез припокриване (overlapping)
- типичната инструкция минава през фазите извличане (от cache - обикновено 1 за цикъл), декодиране (установява функцията за изпълнение и необходимите ресурси - регистри, магистрали, устройства), издаване (резервира ресурсите чрез блокиране и извлича операндите от регистрите към устройствата), изпълнение (1 или повече фази), записване (writeback - на резултатите в регистрите).
- архитектурата на процесорния конвейер Фиг. 2.11
- преподреждане на инструкциите за по-голям коефициент на запълване на цикъла фиг. 2.11.1

- Конвейризацията с ел имитира о тз ависимостта по д инструкциите за преход
- Производителността при програма с 20%/10% вероятност за условен преход между последователните инструкции, 50% вероятност за изпълнение на условието (т.е. на прехода; статистически обаче повечето условни преходи - 60% -се изпълняват) и 8-фазен конвейр е 41%/25% по-малка отколкото производителността при програма, в която поне едната вероятност е 0. Затова при конвейрни процесори е желателно алгоритъмът да се кодира с минимум условни преходи.
- Предвиждането на преходите се използва за да се отложи прехода докато се изпълнят опр. брой инструкции, независими от условието на прехода. То може да бъде базирано на кода на програмата - статично или на историята на изпълнението - динамично

CISC

- класическа архитектура (първите процесори са ограничен набор инструкции)
- уверения набор инструкции настъпва с микропрограмирането с промяната на SWcost/HWcost (първосигнална реакция на семантичната ножица между HLL и процесорните архитектури/машинните езици)
- параметри:
 - 120 – 350 инструкции с няколко формата на инструкциите и данните
 - 32 – 64 регистъра с общо предназначение
 - 4 – 16 режима на адресиране
 - голяма част от изразите на HLL са микрокодирани (т.е. имат съответствие в набора инструкции)
- скаларни CISC процесори – за операции върху скаларни данни; частична конвейризация поради:
 - зависимост по данни между последователните инструкции
 - ресурсен конфликт
- фиг. 2.14.

Разграничават се класовете RISC и CISC по следните параметри:

- формат на инструкцията и на данните
- режими на адресация
- регистърно адресиране (регистри с общо назначение)
- управление на изпълнението на програмата (flow control instructions)

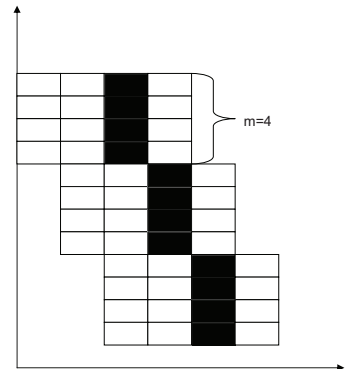
RISC

- 25% от машинните инструкции кодират 90% от HLL програмата и се изпълняват 95% от процесорното време
- подходи за оптимизация:
 - трансформиране на микропрограма памет в регистърен cache
 - FPU и други специализирани устройства на процесорния чип
 - суперскаларни процесори
 - броя на инструкциите е < 100 – с ф и к с и р а н ф о (предвидно регистър-регистър)
 - до 5 режима на адресиране, инструкциите са предимно от тип load/store
 - “регистърни фалове” – по 32+ вътрешни регистри за бързо превключване между процесите
 - едночипови затова висока тактова честота CR и инисък CPI т.е. висок MIPS коефициент
 - скаларните RISC процесори са подобни на скаларните CISC но при еднаква тактова честота производителността може да е по-ниска поради по-малката пълнот на кода
 - необходимост от ефективен компилатор за постигане на високо ниво конвейризация на ниво инструкция
 - суперскаларна RISC архитектура – фиг. 2.15.

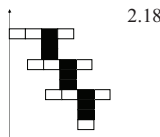
Показатели	CISC	RISC скаларен
Брой инструкции	128-256-300	24-32
Формат на инструкции	16-64 бита, т.е. инструкцията е с плаваща дължина	32 бита, т.е. дължината е фиксирана
Формат на адреси	8-12 бита, различни начини на адресиране на операционната памет, къси/дълги	Регистър-регистър (много по-голям брой вътрешни регистри), регистърни файлове, 3-4 броя на регистърните формати
CPI брой процесорни тактове	8-20 процесорни такта, т.нар. инструкции с различна степен на сложност	3-6 процесорни такта, инструкциите са с фиксирана дължина – опростени
CM управляващ контролен модул	Базира се на микропрограмиране	С помощта на апаратна логика (AJI) hardware control

Суперскаларни процесори (RISC и CISC)

- Повече от 1 инструкция на такт поради наличието на няколко (напр. 3) инструкционни конвейера – съответно няколко резултата от всеки инструкционен цикъл
- модел MIMD инструкционно ниво
- разлика от векторните процесори, които реализират SIMD на инструкционно ниво
- паралелизма се реализира на инструкционно ниво – само между логически независими инструкции
- кратност на инструкцията $m = 2$ до 5 (при скаларните процесори $m = 1$)
- суперскаларен RISC процесор – фиг. 2.17.



VLIW процесори



- Комбинират концепцията за хоризонтално микрокодирани и суперскаларна архитектура:
- дълги инструкции (стотици битове), които задават по няколко операции над операндите
 - различават се от суперскаларните процесори по
 - бързо и просто декодиране на инструкциите понеже една VLIW инструкция замества няколко суперскаларни
 - ниска пълнот на кода но по-висок паралелизъм на инструкционно ниво (понеже инструкциите са с фиксиран формат който може да включва и неизпълними операции, а суперскаларните операции са само изпълними)
 - непреносим обектен код понеже нивото на паралелизма при различните процесори е различно а е заложено в самата дълга инструкция (суперскаларните архитектури са портабелни със скаларните) – само за специализирани компютри
 - инструкционния паралелизъм се задава на етапа компилация – т.е. статичен; няма динамична диспечеризация и синхронизация
 - VLIW процесор – фиг. 2.18.

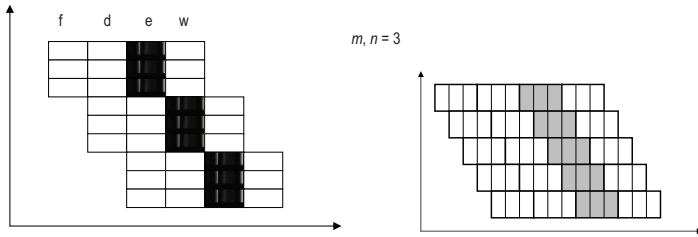
Векторни процесори

- Специализирани копроцесори за векторни операции – операндите в отделната инструкция са масив[n]
- дългите вектори (надвишаващи дължината на регистърните файлове) се сегментират
 - инструкциите са тип
 - регистър-регистър – кратки; адресират се регистърни файлове
 - памет-памет – дълги (защото съдържат адреси от основната памет); те могат да обработват по-големи масиви с различна дължина
 - типични векторни операции
 - зареждане на вектор от паметта на компютъра: $V_1 \leftarrow M_n$;
 - запис: $V_1 \rightarrow M_n$;
 - ескаларне: $S_1 \bullet V_1 \rightarrow V_2$;
 - векторна операция, при която и двете операнди са вектори и резултата е вектор: $V_1 \bullet V_2 \rightarrow V_3$;
 - редукция от векторни операнди и резултат скалар: $V_1 \bullet V_2 \rightarrow S_1$;
 - зареждане вектор-вектор: $\bullet V_1 \rightarrow V_2$;
 - редукция на единичен вектор: $\bullet V_1 \rightarrow S_1$;
 - аналогични инструкции от тип памет-памет – операндите са от вида $M(1 : n)$

Суперконвейърна архитектура

2.20

- При степен n цикъла на суперконвейера е $1/n$ от базовия цикъл на фазите. Фиг. 2.20.
- Закъснението за една операция е равно на базовия цикъл, но ILP е n .
- $T(1, n) = k + (N-1)/n$
- $S(1, n) = \dots \rightarrow n$ за $N \rightarrow \infty$.
- Cray1: $n=3$.



2.22

Технологии на процесорите

- Суперскаларните архитектури са по-подходящи за паралелизъм по данни - многократни операции се изпълняват конкурентно на няколко еднотипни устройства (б л о к о в е з а и з п ъ л н я в а т п ъ р т о в е к ъ м регистрови файлове ...). Затова необходимост от по голяма интеграция в чипа – CMOS технология. Комбинират се с RISC архитектура на процесорната дума.
- Суперконвейерите се базират на паралелизъм по управление, поради което съществено при тях е прилагането на устройства с висока тактова честота – т.е. TTL технология.

...Intel Pentium

2.24

- за ефективно съчетаване на работата на конвейерите (т.е. за избягване на някои от случаите на конфликт) работата на двата инструкционни конвейера е «дефазирани» със стъпка от 1 фаза (първата фаза «извличане»)
- със същата цел Dcache е с двупортова организация – по един самостоятелен порт за всеки от инструкционните конвейери
- cache буферите са с асоциативна организация на достъпа: асоциативната памет има 32-байтов TLB с последните адреси така че търсенето на зарежданата страница става в 32 адреса (вместо 8K) планирането на активните страници в cache е подисциплината LRU
- изискването за свързаност (кохерентност) между данните в cache и в ОП се постига чрез специален протокол – MESI – което позволява изграждането на мултипроцесорни архитектури (т. нар. симетричен мултипроцесинг – хомогенна мултипроцесорна архитектура с обща памет между процесорите)
- интегрирано FPU устройство с 8-фазов конвейер (извличане, декодиране, адресна генерация изпълнение, обработка мантиса, обработка експонента, обработка приближение и запис който може да изпълнява и две FP инструкции едновременно (когато едната от тях е присвояване).

Intel Multicore

2.26

- Intel Core Microarchitecture технология прилага интегриране на машинната архитектура на симетричния мултипроцесинг в микропроцесор
- суперконвейърни суперскаларни ядра: 14 фази (коэф. на супреконвейърност 2); по 4 инструкции
- Две/четири независими ядра - NUMA мултипроцесинг с локализиран L1-cache за всяко ядро и общ L2-cache
- елементи на VLIW и RISC (едновременно!):
 - поддържа x86 CISC набор инструкции, като декодира част от тях до 2 и повече конвейърни микроинструкции - RISC
 - т. нар. микрооперации обединяват няколко често срещани последователности от машинни инструкции за изпълнение като една инструкция - напр. проверка на стойност и преход по флаг са обединени в «микрооперацията» условен преход
- интегриран арбитраж на достъпа до L2-магистралата
- разширени възможности на управление на енергийното потребление и версии

Суперконвейърна суперскаларна архитектура

2.21

- Степента е (m, n) като m е кратността на едновременно издаваните инструкции (т.е. на супрескаларност) а n е кратността на супреконвейера ($1/n$ от кратността на базовия цикъл между групите последователни инструкции). Фиг. 2.21. Закъснението за една операция е равно на базовия цикъл, но ILP е n .
- $T(m, n) = k + (N-m)/mn$
- $S(m, n) = \dots \rightarrow nm$ за $N \rightarrow \infty$.
- DEC Alpha: $n=6, m=2$.

Intel Pentium

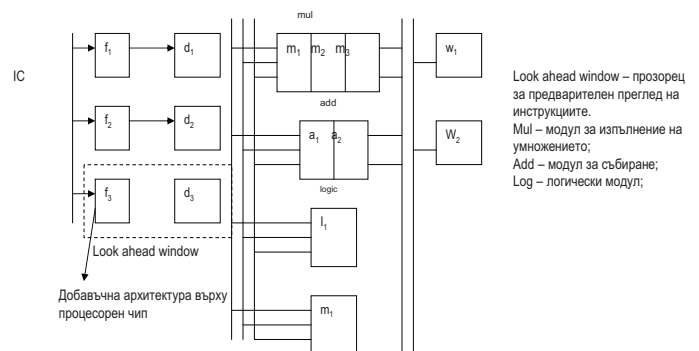
2.23

С въвеждането на Пентийм архитектурата (1995) Интел прилага предимствата на неклассическа паралелна архитектура в производството на процесор, предназначен типично за масови компютри:

- суперскаларен процесор с ниво на инструкционния паралелизъм $m=2$ (3 за P4) – едновременна обработка на 2 целочислени операнда по модела MIMD (когато последователните инструкции нямат зависимост по данни или управление!);
- всеки инструкционен конвейер се състои от 5 фази: извличане, декодиране, адресна генерация (типичен CISC процесор с много режими на адресация на ОП), изпълнение и запис
- изпълнението на последователни инструкции от всеки конвейер е със закъснение 1 фаза (извличане): два самостоятелни I cache + D cache по 8 КБ

...Intel Pentium

2.25



Добавъчна архитектура върху процесорен чип

Look ahead window – прозорец за предварителен преглед на инструкциите.
Mul – модул за изпълнение на умножението;
Add – модул за събиране;
Log – логически модул;

3. Паралелна обработка

3.1

Съдържание

- ➔ Паралелни процеси
- ➔ Паралелни алгоритми – принципи, проектиране, метрика
- ➔ Среди и езици за паралелни програми
- ➔ Синхронизация на паралелните процеси
- ➔ Балансиране на паралелната обработка
- ➔ Еталонни паралелни алгоритми

Последователни и паралелни програми 3.3

- ➔ програмата се състои от процеси, които могат да бъдат изпълнявани последователно или конкурентно
- ➔ при изпълнение на програма в **среда за последователното програмиране**
 - ➔ програмата се състои от един процес
 - ➔ резултатът от изпълнението ѝ с еднакви данни е винаги един и същ
 - ➔ изпълнението на всяка инструкция е последователно и независимо от изпълнението на други инструкции
- ➔ при изпълнение на програмите в **среда с мултипрограмиране**
 - ➔ програмата се състои от един процес
 - ➔ управлението се предава последователно между различни процеси
 - ➔ между отделните процеси съществува зависимост по време на изпълнение, но резултата от изпълнението им се запазва
- ➔ при изпълнение на програмите в **среда за паралелно програмиране**
 - ➔ програмата се състои от множество паралелни процеси
 - ➔ тя включва освен управляващ код и данни, също и инструкции за синхронизация и обмен между процесите, които съставляват нейния планиращ процес (scheduler)
 - ➔ резултатът от изпълнението на паралелната програма може да зависи от работата на планиращия процес

Паралелни процеси

- ➔ процесите, изпълняващи програмата в средите за паралелна обработка, могат да бъдат алтернативно:
 - ➔ реплики, изпълняващи еднакви подпрограми върху различни данни – модел **SPMD (Single Program Multiple Data)**. **N.B.**: разликата от **SIMD** е, че в този случай синхронизацията се извършва на ниво подпрограма (сегмент), а не на ниво инструкция и затова **SPMD** обслужване се изпълнява на **MIMD** компютри
 - ➔ различни подпрограми – модел **MPMD (Multiple Program Multiple Data)**; при този подход отделните подпрограми-процеси се пораядат като дъщерни на един [главен] процес

Граф на процесите ([precedence | dependency] graph)

- ➔ зависимостта по данни и управление се изследва [чрез графи] на различни нива – блок, израз, променлива
- ➔ компилаторите обикновено изследват графа на зависимостите на ниво израз и променлива – пример за серията изрази (фиг. 2.5):

$$S1: A = B + C$$

$$S2: B = A + E$$

$$S3: A = A + B$$
 - изразите се изобразяват като възли в графа на зависимостите, а дъгите са зависимостите като началото на дъгата е променлива (аргумент или стойност) на израз, а край – същата променлива от следващ израз – освен когато началото и края на дъгата са аргументи (от дясната страна) на изразите

Типове зависимости в графа на процесите 3.6

- ➔ **зависимост по данни (data flow)**: резултата от израз е аргумент на следващ израз (пренареждането на изразите или паралелното им изпълнение променя резултата на следващия израз – вж. упражнението 1 за примера от т. 5. и други примери) – тази зависимост е непреодолима
- ➔ **антизависимост (anti-dependency)**: аргумента на израз е резултат от следващ израз (пренареждането на изразите или паралелното им изпълнение променя резултата на анализиращия израз) – тази зависимост може да бъде преодоляна чрез репликиране на променливите
- ➔ **зависимост по изход (data output)** – резултатите от два израза се записват в една и съща променлива (пренареждане или паралелно изпълнение променя стойността на тази променлива) – тази зависимост може да бъде преодоляна чрез репликиране на променливите
- ➔ **зависимост по вход (data input)**: два израза имат общ аргумент – тази зависимост няма значение при съвременните програмни системи (поради средствата за конкурентен достъп)
- ➔ **зависимост по управление (data control)**: условно изпълнение на израз, където условието е резултат от предходен израз (по същество това е разновидност на зависимостта по данни)
- ➔ за по-висок паралелизъм на кода се отстраняват антизависимостите и зависимостите по изход

Пример за отстраняване на зависимости

изходен код

```
for i = 1, n, 1
  x = A[i] + B[i]
  Y[i] = 2 * x
  x = C[i] * D[i]
  P = x + 15
endfor
```

код с намалена зависимост

```
for i = 1, n, 1
  x = A[i] + B[i]
  Y[i] = 2 * x
  xx = C[i] * D[i]
  P = xx + 15
endfor
```

Модели обща памет 3.8

- ➔ в паралелните системи достъпът до общата памет и ресурси за В/И е конкурентен и се базира на схемите за **PRAM (Parallel Random Access Machine)** – автономни процесори с конкурентен достъп до обща памет (която включва и В/И канали)
- ➔ в модела **PRAM** се предлагат 4 схеми за отстраняване на конфликтен конкурентен достъп до общото адресно пространство:
 - ➔ **EREW (Exclusive Read, Exclusive Write)** – резервиране на конкурентния достъп да даден адрес за двата типа операции
 - ➔ **CREW (Concurrent Read, Exclusive Write)** – няколко процесора могат да четат едновременно даден адрес, но операциите за запис са монополни
 - ➔ **ERCW (Exclusive Read, Concurrent Write)** – допускат се няколко едновременни операции на запис но монополно четене
 - ➔ **CRCW (Concurrent Read, Concurrent Write)** – конкурентните операции са без ограничение
- ➔ ****EW** схемите съответстват на изискванията за консистентност (съгласуваност и детерминистичност) на данните и се прилагат като универсални при повечето паралелни алгоритми;
- ➔ конкурентните операции за запис при ****CW** схемите имат ограничено приложение при някои класове паралелни алгоритми за обработка на графи и числова обработка, при които постигат по-високо бързодействие от схемите с резервиран запис

Модел с обмен на съобщения

- ➔ при обмен на съобщения всяка двойка процеси е свързана с комуникационен канал, представен с точно една променлива – последователните съобщения са стойностите на тази променлива;
 - ➔ дефинирано е състояние на канала – напр. четене на променливата-канал се допуска само когато състоянието му не е празен (респ. при запис – да не е пълен);
 - ➔ асинхронният и синхронният канал са с еднакъв режим на достъп но асинхронният има капацитет = размера на буфера (>1)

Паралелни алгоритми

- Паралелните алгоритми са междинното звено във веригата на паралелната обработка (между изчислителния проблем и паралелната система) –
 - архитектура
 - система/среда
 - програма
 - алгоритъм
 - изчислителен проблем
- ПА е абстрактно (формално или неформално) представяне на изчислителен проблем като набор от процеси за едновременно изпълнение (в случая процес е част от проблема, която се изпълнява от един процесор)
- основните характеристики на ПА (които отсъстват при посл. алгоритми) са
 - брой процеси и логическата им организация (напр. master-slave)
 - разпределение на данните (декомпозиция + възможности за разпределена алокация)
 - точки на синхронизация (оптимизиране)
 - модел на междупроцесния обмен (основно обща памет – обмен на съобщения)
- различните конкретни решения на горните характеристики порождават цял клас от ПА, базирани на един последователен алгоритъм

3. Паралелна обработка

ФМИ/СУ * ИС/СИ * РИТАрх/РСА

10

... фази на проектирането на ПА

3.12

- формиране (agglomeration) – след оценка на изчислителната и комуникационната сложност на формулираните подзадания и прилежащите им комуникации, те се групират в задания, при което се отчитат характеристиките на архитектурата на обработка – основно брой процесори/възли и комуникационен модел – и в резултат се постига оптимизиране по следните характеристики
 - грануларност и балансираност (с оцека на изчислителната сложност на отделните задания)
 - евентуално репликиране на данни и подзадания
 - оптимизиране на комуникациите (с оцека на комуникационната сложност на отделните задания)
 - евентуално запазване на линейност (скалируемост)
 - технологично оптимизиране (напр. намаляване на разходите за кодиране на заданията)

3.14

Метрика и анализ на производителността

- сложността на последователните алгоритми (брой операции) се оценява като функция само на размера на проблемната област и следователно може да се оцени абстрактно от архитектурата; при ПА тя е функция на архитектурата и на средата за паралелна обработка (особено при динамично планиране)
- основен фактор при ПА е степента на паралелизъм P – максималния брой операции, които могат да се изпълнят паралелно при обработката на алгоритъма – това е архитектурнонезависима величина; при размер на проблема W не повече от $P(W)$ процесора могат да се ползват ефективно; съществено е съотношението между паралелните и последователните сегменти на ПА

3.16

Ускорение и ефективност

- при оценка или измерване на ускорението ($S_p = T_1/T_p$) се приема, че всички процесори в двата случая са с идентична производителност; поради наличие на комуникационни и синхронизационни закъснения $1 < S_p < p$
- аномалии:
 - **суперлинейно** $S_p > p$ може да се наблюдава при
 - неоптимален последователен алгоритъм или
 - особени характеристики на проблема, които изявяват нисък капацитет на използвания хардуер: напр. при голям размер на данните (надвишаващ капацитета на ОП) е възможно значително закъснение на последователната обработка на проблема поради бавни операции с външната памет, докато при паралелна обработка с разделянето на данните между възлите този проблем отпада (оптимизиране на последователния алгоритъм в този случай не е възможно когато за данните не важи принципа на локалност – напр. при много проблеми от AI)
 - **немонотонно** $S_{p_1} > S_{p_2}$ за $p_2 > p_1$ – често срещана аномалия
- ефективността, която е нормализирано ускорение ($E_p = S_p/p = T_1/(pT_p) < 100\%$), характеризира частта от общото време за паралелна обработка, през която процесорните елементи се използват

Фази на проектирането на ПА

3.11

- проектирането на ПА минава през следните фази (4.3):
 - разделяне (partitioning) – декомпозиция на проблема:
 - по данни (главно SPMD) или
 - по функции (главно MPMD) –
 - разделянето се извършва с оглед на спецификата на проблема; целта е да се дефинират множество подзадания; грануларността при тази фаза не отчита особеностите на архитектурата, която ще се използва за обработка – резултатът от фазата е дефиниция на отделните задания
 - комуникации (и зависимост) (communication) – формулира информационните или контролните зависимости между отделните подзадания; комуникациите се представят като канали (със съответните свойства – напр. капацитет, посока) и съобщения (т.е. данни и команди), които се предават по тези канали (напр. формат, размер, тип); архитектурата за обработка се игнорира и на тази фаза, но специфицирането на каналите помага да се оцени алгоритъма по комуникационна сложност

... фази на проектирането на ПА

3.13

- разпределяне (mapping) – незадължителна фаза (отсъства при проектиране на ПА за системи с динамично планиране – обикн. МП с РОС), която се състои в разпределяне на формираните задания (или евентуално групи от задания) по обработващите възли на системата със кодиране на съответното решение. **N.B.:** обикновено се използва специален език за спецификация на зареждането и евентуално за настройка на комуникационните канали напр. в системи с комутируеми канали, така че от алгоритъма се изисква да специфицира и комуникационния граф на системата за обработка 5.1

3.15

Закон на Amdahl (1967):

- при наличие на две интензивности (темпове) на обработка на даден порблем – високо-паралелна R_h и ниско-паралелна R_l , които са в съотношение $f: (1-f)$ по брой на генерирани резултати (междинни и крайни) – общата интензивност на обработка е

$$R(f) = [f/R_h + 1-f]/R_l^{-1}$$
- следователно $f \rightarrow 1$ $R(f) \rightarrow R_h$ и при $f \rightarrow 0$ $R(f) \rightarrow R_l$
- **N.B.:** макар че е формулиран за темпове на обработка, закона е в сила и се прилага за агрегирана степен на паралелизма на заданието

3.17

Пример за оценка на ускорението и ефективността

- хиперкуб от p процесора изчислява сумата на n числа; времето за локална операция събиране на две числа и времето за предаване резултата на съседен процесор е 1
- ПА: числата се сумират локално за време n/p след което локалните (първоначално p на брой) парциални суми се предават на съсед (1) и сумират (1) за $2lbp$ (4.5.1)
- $S_p = n/[n/p + 2lbp] = np/[n + 2plbp]$; $E_p = n/[n + 2plbp]$
- получените зависимости (4.5.2) показват обичайният ефект при по-голямата част от ПА на намаляване на ефективността с нарастване на p (при фиксиран размер на проблема n) – ефект от закона на Amdahl

Цена и коефициент на използване

- цена (cost) при обработката на ПА с p процесора за T_p единици време (N.B. единица време е времето за изпълнение на една елементарна операция) е

$$C_p = pT_p$$
- т.е. C_p е критерий за броя операции, които биха могли да се извършат за времето на обработка на съответния ПА
- коефициент на използване (utilization) при обработката на ПА, състоящ се от O_p на брой операции с p процесора е

$$U_p = O_p / C_p = O_p / (pT_p)$$

- т.е. U_p е отношението на действителните към потенциалните операции при обработка на съответния ПА

3.20

Цена и коефициент на използване

- цена (cost) при обработката на ПА с p процесора за T_p единици време (N.B. единица време е времето за изпълнение на една елементарна операция) е

$$C_p = pT_p$$
- т.е. C_p е критерий за броя операции, които биха могли да се извършат за времето на обработка на съответния ПА
- коефициент на използване (utilization) при обработката на ПА, състоящ се от O_p на брой операции с p процесора е

$$U_p = O_p / C_p = O_p / (pT_p)$$
- т.е. U_p е отношението на действителните към потенциалните операции при обработка на съответния ПА

3.22

Паралелно програмиране в разпределени системи

- прилага моделите
 - разпределена обща памет (DSM)
 - ключалки семафори, монитори, бариери
 - обмен на съобщения (Message Passing Systems)
 - приложно-ориентиран междинен слой –
 - MPI и PVM – процедурен модел
 - RMI и Corba – обектен модел
 - йерархични (master-slave, client-service - Jini) и n е йерархични модели (P2P - Jxta)

3.24

Блок FORALL

- този блок се прилага за имитация на паралелно изпълнение на вложения в него сегмент (набор изрази) – асинхронно (независимо – напр. в MIMD) или синхронно (напр. в SIMD)
- синтаксис:


```
FORALL identifier: RangeType IN {PARALLEL | SYNC}
  Statement_1
  ...
  Statement_K
END
```
- identifier е управляваща променлива, дефинирана в границите на блока; по един процес се създава за всяка нейна стойност (множеството стойности трябва да е крайно); в създадените процеси identifier има различни стойности
- RangeType е типът на управляващата променлива, чиято мощност освен това задава и броя паралелни процеси
- изпълнението на блока завършва след изпълнение на всеки от процесите
- PARALLEL или SYNC задава типа паралелна обработка – съотв. асинхронен или синхронен (асинхронната обработка означава, че част от процесите могат да се планират след изпълнението на другите – напр. когато броят им е по-голям от броя процесори)

Темп и излишък

- темпът на обработка (execution rate) е архитектурнозависим параметър и се представя с няколко скали:
 - MIPS (унипроцесори, МП)
 - MFLOPS (SIMD, числова обработка)
 - MOPS (SIMD)
 - LIPS [# logic inferences p.s.] (AI приложения)
 - освен по архитектурен критерий, изборът на скала зависи и от типа ПА, които се обработват
- излишък (redundancy) при обработката на ПА, състоящ се от O_p на брой операции при обработка с p процесора е

$$R_p = O_p / Q_1 > 1$$
 (където Q_1 е броя операции при обработка на униккомпютър) т.е. R_p е критерий за свръхтовара, който се поражда от паралелната обработка на алгоритъма; p и n (размера на проблема) са аргументи на R_p но – в зависимост от класа ПА – участват с различна тежест

3.21

Алгоритмична сложност

- коректността на даден ПА е архитектурнонезависима, но неговата ефективност зависи от изпълнителната платформа, поради което е целесъобразно сложността му да се оценява и като функция на разпределянето (mapping)
- по принцип алгоритмичната сложност O оценява времевите и пространствени характеристики на обработка – времевата сложност T е се задава в брой елементарни операции и комуникации (от които се получава времето за обработка в дадена архитектура), а пространствената сложност M в брой алоцирани регистри и клетки памет (т.е. $O = O(T, M)$); оценката се дава обикновено като долна и горна граница на тези величини или с приближение – асимптотична сложност

3.23

Конвенционален псевдокод за паралелни алгоритми

- псевдокодът (както и езиките за програмиране) е приложим за определени класове архитектури – обикновено се взима като предпоставка най-разпространения PRAM модел за паралелен достъп до обща памет (променливи) – CREW
- декларация на процедури и функции е разширена със запис на модела за паралелна обработка и броя алоцирани процесори:


```
Procedure: <name> ({list of parameters})
Model: <model name> with p = f(n) processors
Input: <input variables>
Output: <output variables>
Declare: <[global and] local variables>

Function: <name> ({list of parameters})
Model: <model name> with p = f(n) processors
Input: <input variables>
Output: <output variables>
```

3.25

Пример за блок FORALL

- 8 процеса за асинхронна паралелна обработка на функция с аргумент – номера на процеса


```
FORALL x: [1..8] IN PARALLEL
  y = some_function(x);
END
```
- версия


```
FORALL x∈X IN PARALLEL do y = some_function(x);
```

Израз do IN PARALLEL

3.26

- този израз се прилага като директива в различни блокове – напр. при паралелна векторна обработка
- синтаксис:

```
for <израз върху индексите на масив> do IN PARALLEL
  Statement_1
  Statement_2
  ...
  Statement_K
end IN PARALLEL
```
- пример: за всеки елемент на масивите се формира отделен процес

```
for i = 1 to n do IN PARALLEL
  read(A[i], B[i])
  if (A[i] > B[i])
    then write(A[i])
    else write(B[i])
  endif
end IN PARALLEL
```

3.28

Синхронизиращ псевдокод със semaфор

```
P1: wait(S1)
   {critical section 1}
   signal{S1}
P1: wait(S1)
   {critical section 2}
   signal{S1}
```

3.30

Синхронизация с бариери

- с бариерите се осъществява контрол за последователност – напр. за запазване на зависимостта по данни
- бариерата също се състои от буфер за готови изчакващи процеси и боряч
- псевдокод с използване на бариера:

<i>Псевдокод без синхронизация</i>	<i>Псевдокод с бариерна синхронизация</i>
<pre>For I = 1 to N do IN PARALLEL { S1: A[I] = func_a(A[I]) S2: B[I] = func_a(B[I]) S3: C[I] = func_c(A[I], B[I]) }</pre>	<pre>For I = 1 to N do IN PARALLEL { S1: A[I] = func_a(A[I]) S2: B[I] = func_a(B[I]) BARRIER(2) S3: C[I] = func_c(A[I], B[I]) }</pre>

3.32

Статично балансиране

- разпределянето на заданията по възли и алоцирането на ресурси се извършва (и е известно) преди да стартира паралелната обработка – планиране, комплементиране (**mapping, matchmaking, scheduling**)
- подходи за статично балансиране
 - RR – циклично алоциране на заданията по обработващи процеси
 - стохастично разпределяне
 - рекурсивно разделяне – при алгоритмите за графи – бисекция (разделяне на проблема на подпроблеми с очаквана еднаква сложност на обработка и с генериране на минимален синхронизационен и комуникационен свързтовар)
 - генетични и Монте Карло алгоритми – свързани са с генериране на възможни варианти на декомпозицията и оценяването им, така че да се избере оптималния

Синхронизационни конвенции, semaфори

3.27

- синхронизационните схеми биват
 - контрол на достъп – semaфори и монитори
 - контрол за последователност – бариери
- променлива от тип semaфор се асоциира с всеки адрес за общ достъп и върху нея се извършват операциите
 - установяване на състоянието (активно или пасивно) (`wait`)
 - блокиране на процес (`wait`)
 - възстановяване от блокиране (`signal`)
- `wait(S)` е заявка за достъп до критичната зона, която се потвърждава ако $S > 0$ (и S се декрементира); в противен случай процеса блокира и изчаква
- `signal(S)` освобождава критичната зона, инкрементира S и възстановява чакащ процес

3.29

Синхронизация с монитори

- мониторите са разширение на semaфорите, което се състои както от данните за контрол на достъпа – **condition variable**, така и от процедурите – **signal** и **wait**
- при дефиниране на **condition variable** се създава и опашка на идентификаторите на чакащи процеси, които се възстановяват и получават достъп до критичната зона с операцията **signal**

```
Monitor Resource_alloc      ...
Var Resource_in_use: Boolean; Procedure Release_resource
    Resource_is_free: Condition; begin
Procedure Get_resource      Resource_in_use = false
begin                        signal(Resource_is_free)
    is (Resource_is_free) then end
    wait(Resource_is_free)    end Monitor
    Resource_in_use = true
end
```

Задачи на балансирането на изчислителния товар (Load Balancing – LB, Resource Management, Resource/Job Scheduling)

3.31

- минимизиране времето за решаване на даден проблем при паралелна обработка чрез изравняване на локалното натоварване на обработващите възли
- целта може да бъде не пълно изравняване а недопускане на възел в престой докато трае паралелната обработка
- в г р и дпропорционално натоварване на ресурси с различна собственост и администрация
- източници на дисбаланс
 - нерегулярност на проблема при паралелизъм по данни
 - недетерминистични алгоритми за обработка, напр. при неизвестен брой итерации за достигане до решението – търсене в графи и др.
 - невъзможно или некомпетентно декомпозиране – при паралелизъм по данни или по управление

3.33

Недастатъци на статичното балансиране

- проблемна предварителна оценка на сложността на подпроблемите, получени при декомпозицията
- не може да отчете текущото състояние на ресурсите по време на обработката – фоновото натоварване на ресурсите (процесорни цикли, памет, комуникационни канали) както и реалните синхронизационни и комуникационни закъснения – ограничено приложение аз синхронни алгоритми
- при недетерминистични алгоритми за обработка, напр. при неизвестен брой итерации за достигане до решението – търсене в графи и др. – статично решение на задачата за товарен балнс е невъзможно освен чрез прилагане на по-фина грануларност и откриване на край (**distributed termination detection**)

Динамично балансиране

- разпределянето на заданията по възли и алоцирането на ресурси се извършва по време на паралелната обработка и е известно едва след приключването ѝ
- централизиран подход – **master-slave** обработка; декомпозицията, разпределянето на заданията и ресурсите, откриването на край или алтернативно интегрирането на резултата са функции на един **master** процес
- разпределен подход – декомпозиция на управляващия процес в йерархия от управляващи процеси или асоцииране на управляващите функции с всеки от обработващите процеси

Разпределено динамично балансиране

- пряк подход е разпределяне на функциите на управляващия процес по поддържане на динамичния пул от задания на йерархичен слой но управляващи процеси – фиг. 5.6.
- оптимизацията в горния случай е предимно в избора на брой управляващи процеси от втори ниво или евентуално избор на броя управляващи нива
- при някои алгоритми се практикува развито йерархично дърво – обикновено двоично тъй като разделянето на [под-]проблема на две очаквано равно части е по-лесно за алгоритмиране и за прилагане на рекурсия

Параметри на P2P динамичното балансиране

- подобни балансиращи схеми се наричат дифузионни, тъй като реализират балансирането чрез трансфер на под-задания към "съседни" възли; релацията за съседство в случая може да изхожда от конкретна топология на изпълнителната платформа, но може да бъде и подчинена на различни стохастични принципи – напр. на случаен избор от определен брой (оптимизационен параметър!) "съседни"
- в горния случай като средство за повишаване на линейността на алгоритъма се избягват схеми когато всички възли са "съседни"; вместо това се формират виртуални топологични структури – линия, пръстен, хиперкуб и др. (обикн. неийерархични) топологии; когато валентността на процесите е по-голяма от 1, може да се прилага циклично или случайно търсене на "съсед" за балансиращ трансфер
- друг важен параметър на P2P балансиране е инициативата (или момента за активиране на локалната балансираща процедура):
 - инициатива на донора
 - инициатива на приемника

Еталонни паралелни алгоритми

- асинхронни алгоритми – **Mandelbrot set**
- локално-синхронни алгоритми – **Water simulation, odd-even sort**
- глобално-синхронни алгоритми – **n-body simulation, Ray tracing**

Централизирано динамично балансиране 3.35

- главния процес функционира като пул от задания (**work pool**) и получава заявки за ново задание от готовите изпълнителни процеси; изпълнителните процеси са обикновено реплики (модел **SPMD**)
- пултът от задания се прилага при матричните изчисления, при алгоритмите "разделяй и владей"
- нерегулярните и динамичните товари също са подходящи за **work pool** обработка – в последния случай генерираните от обработката нови задания се присъединяват в опашката на пула заедно с текущия резултат от изпълнителния процес – фиг. 5.5.
- основно предимство на централизираното динамично балансиране е лесното установяване на изпълнение на условието за край – при празен пул и прекратена работа на изпълнителните процеси; при някои алгоритми за търсене условието за край се открива от някой от изпълнителните процеси и се предава към главния процес заедно с резултата
- недостатък е възможността за възникване на тясно място и ниската линейност

P2P динамично балансиране

- т о е ф о р м а н а п ъ л н о прилагане на разпределеното динамично балансиране. Премахва се разделението на управляващи и изпълнителни процеси като всеки процес извършва и двете функции
- формално и опростено цялото задание може да бъде предадено за изпълнение в един процес/възел, след което се извършва неговата декомпозиция и последващ балансиращ трансфер на генерираните подзадания между възлите
- в този случай декомпозицията е желателно да бъде или тривиална (примерно при матрични изчисления) или пък да бъде опростена (примерно бисекция на проблема без първоначален анализ колко са потенциалните обработващи процеси, какво е тяхното текущо натоварване и каква е оптималната грануларност)

Системи за динамично балансиране

- информационна, локационна и трансферна стратегия
 - функции
 - граф
 - разпределение
- синхронно балансиране – **co-scheduling**
 - **Koala** (<http://www.omii.ac.uk/repository/project.jhtml?pid=122>, & <http://www.cs.vu.nl/~kielmann/ascii-a14/slides/koala/koala.pdf>)
- асинхронно балансиране – **htc** (High throughput computing), **volunteer computing**
 - **Condor/Condor-G** (<http://www.cs.wisc.edu/condor/>), **Boinc** (<http://boinc.berkeley.edu/>)
 - балансира се нископриоритетните процеси на опортюнистичните потребители – във фонов режим (background priority)

кълъстерно, мултикълъстерно и с2с планиране

4. Модели на разпределена софтуерна архитектура

Съдържание

- Модели софтуерна архитектура
- Спецификации с UML
- Структурни и функционални функционални диаграми
 - Модели на изгледи
 - Спецификации с ADL

Модели софтуерна архитектура

- Софтуерната архитектура представя – т.е. моделира – програмния проект (процес на обслужване) като съставен т.е. разпределен процес от софтуерни компоненти
- моделирането на PCA е първата и най-важна фаза на проектиране, настройка, тестване, разгръщане и документация на разпределени среди за обслужване
- моделът на дадена софтуерна архитектура описва
 - декомпозицията на процеса на компоненти
 - функционалната им композиция
 - прилагания архитектурен стил – напр. процедурен, обектен, потоков (data flow), йерархичен или не-йерархичен, информационен (data centric), интерактивен (interaction oriented), базиран на изгледи (views) и др
 - качественте (нефункционалните) атрибути на услугата – QoS

Представяне на софтуерните модели

- Използват се графи и техни разширения
- описанието е чрез диаграми или техни текстови еквиваленти
- цели на описанието са
 - визуализация
 - спецификация
 - конструиране
 - документация
 ☞ следователно обикновено моделът включва мн. повече от една диаграма
- описанието (моделирането) стартира от по-упростените концепции на бизнес-модела или потребителския сценарий
 - напр. едномерен модел с блокова диаграма (ненасочен граф) – 4.4
- за по пълно функционално и нефункционално описание на проекта се
 - напр. «4+1» модели, включващи
 - логически изглед
 - изглед процеси
 - изглед проектиране
 - физически изглед
 - потребителски интерфейсни изгледи

UML-модели на СА

- използва се за OO-спецификация, анализ, проектиране и документиране на софтуерни проекти
- спецификациите са в две групи диаграми:
- структурни диаграми – **статично** описание (изреждане) на лентите в системата
 - йерархична библиотека класове
 - статични връзки между класовете
 - наследяване (“is a”)
 - асоциация (“uses a”)
 - агрегация (“has a”)
 - обмен (method invocation)
- функционални (behavioral) диаграми – **динамично** описание функциите (“поведението”) на инстанциите на класовете (т.е. обектите) с диаграми на
 - колаборацията,
 - акцията и
 - конкурентността между обектите
- UML диаграмите могат да се транслират до HLL с общо приложение

Структурни UML диаграми

Class	Изброяване и статични връзки между класовете (независещи от взаимодействието им по вр. на изпълнение)
Object	Извлечение от клас диаграмата за обектите и тяхното взаимодействие в определени специфични моменти от изпълнението на системата
Composite	Диаграма на съставни структури – описание на структурата на даден компонент като съставляващи го класове и компонентните интерфейси
Component	Описание на системата като структура от компоненти, интерфейсите между тях, и общите системни интерфейси
Package	Йерархична пакетна структура на организацията класовете в иерархии (т.е. групирани файлове) – пакети от класове и пакети от пакети
Deployment	Диаграма на разгръщането - описание на изпълнителната инфраструктура: сървери, изпълняващи компонентите , системно осигуряване и мидълуер, интерфейси и протоколи, вътрешна и външна мрежова свързаност

Функционални UML диаграми...

Use case	Диаграма на случай на употреба – потребителските сценарии на заявки към системата и техните реакции – за описание на функционалните и нефункционалните изисквания към системата
Activity	Диаграма на дейностите – описание на контролния и контекстния обмен между класовете като мрежа от акции, които системата изпълнява за да осъществи реакциите по потребителския сценарий – оркестрация на акциите
State Machine	Диаграма на машината на състоянията – описание на жизнения цикъл на обектите като машина на състоянията – диаграми на състоянията и преходите (активни вътрешно-обусловени и реактивни външнообусловени преходи)

... функционални UML диаграми

Interaction Overview	Диаграма за преглед на взаимодействието – описва потока команди между обектите (control flow) и е комбинация от Action и Sequence диаграмите
Sequence	Диаграма на последователност - нареден (т.е. времеви) списък от съобщенията между обектите
Communication	Аналогично на Sequence диаграмата, но структурирана като комуникационни канали , които съдържат определен брой последователности
Time Sequence	Времево описание на преходите между вътрешните състояния на обектите и на различимите външни събития (от потребителския сценарий) като последователност от съобщения

Class диаграми

- най-разпространеното описание при всеки модел
- статично изброяване на съставните блокове на модела като **класове**
- задава **«речника»** на модела в съответствие с проблемната област
 - тип
 - интерфейс
 - методи
 - свойства
- достъпността (видимостта) на атрибутите се описва като
 - public
 - private
 - protected
 - default
- описва се и отношенията между класовете – наследяване, асоциация, агрегация (чрез дъги)
 - а също и мощността на тези отношения – 1:1, 1:много и т.н. (чрез маркировки в края на дъгите)

Class диаграма - пример

- фиг. 4.10
- система за потребителски заявки
- наследственост (стрелка към родителя/базовия клас)
- агрегация ром към корена
- асоциация (нейерархична дъга)
- маркировка на мощността в двата края на дъгите

Object диаграми

- извлича се от клас-диаграмата
- описва обектите като инстанции на класовете т.е. примерно подмножество обекти за дадена клас-диаграма конкретен момент на работата на системата
- пример – фиг. 4.11

Composite Structure диаграми

- описва връзката между обектите (*runtime*), с което разширява “речника” на модела
- обектите и връзката се анотират с етикети – съответно на ролята изнес- или функционална логика и отношението им (“колаборацията”)
- пример – фиг. 4.12

Component диаграми

- компонентите са изпълними *SW*-модули за многократно използване при проектиране, които се представят със своя интерфейс
- в те са със скрита структура черна кутия но при различните технологии се прилагат и компоненти тип “сива” и “стъклена кутия”]
- напр.
 - *jar* в компонентната библиотека *JavaBean*
 - *dll* в *.Net*
- компонентната диаграма представя съответствието между квантите пукръгче и имплементирани кръгче интерфейси – фиг. 4.13
- компонентите в даден проект може да са готови – *COTS* – и специфични

Packet и Deployment диаграми

- фиг. 4.14.1
- фиг. 4.14.2

Use case диаграми

- описва потребителските сценарии на приложение на системата като граф от актори, случаи на употреба (потребителски функции) и връзките между тях
- акторите са крайни потребители или други системи, приложения и устройства
- случаите (*Use Cases*) са комплексни функционални модули от разпределеното приложение/проекта, които описват отделни стъпки от цялостната бизнес-логика
- описанието на случаите се допълва в други диаграми с пред- и след условията на изпълнението им като последователности от стъпките на общото приложение при конкретно негово изпълнение
- връзките между сценариите (фиг. 4.15) се маркират с
 - <<include>> от случай, който използва друг случай за изпълнение на дадена функция (насочена дъга)
 - <<extend>> от случай, който звиква друг такъв за изпълнение на функция по изключение (т.е. като опция, която се изпълнява само по изключение)
 - диаграмите на случаите на употреба са основа на описанието и [началните] им версии се използват за основа на структурните и *sequence* диаграмите

Activity диаграми

- описват проекта като *потоков* (*workflow*) бизнес процес, състоящ се от дейности – *activities*
- дейностите капсулират
 - логиката на взимането на решение
 - конкурентното изпълнение на функции
 - обработката на изключения
 - прекратяването на процеса (*termination*)
- потоковата *activity* диаграма (фиг. 4.16) се състои от
 - една начална точка и поне една крайна точка (плътен кръг и ограден кръг)
 - точките на решаване (означават се с ромбче)
 - другите дейности (заоблен правоъгълник)
 - събирането на два и повече потока се счита за синхронизатор (следващите го дейности не могат да стартират без завършване на всички предхождащи го)
 - събития (*events* - опция) – представят обмена на съобщения (*signals*) между конкурентните акции (насочени многоъгълници с етикети)

State Machine диаграми

- обикновено представят състоянието на обслужващите устройства или софтуерните модули в проекта – набор от състоянията им и преходите между тях
- логиката на състоянията е реактивна – т.е. базирана на външни събития (*events*)
- състоянията се описват с блок, съдържащ
 - име,
 - списък променливи
 - *activity*
- *State Machine* диаграмата (фиг. 4.17) се състои от
 - една начална точка и поне една крайна точка (плътен кръг и ограден кръг)
 - насочени маркирани дъги на преходите
 - състоянията, които може да са комплексни състояния, съставени от допълващи *State Machine* диаграми

Interaction Overview, последователности и времеви диаграми

4.18

- диаграмите за преглед на взаимодействието се състоят от кадри (frames), които представляват други диаграми на проекта, маркирани с указател (reference) или със самите диаграми, маркирани с тип – напр. sd, cd, ad
- дъгите отразяват контролния поток на взаимодействието - фиг. 4.18.1
- sequence диаграмите отразяват относителната последователност от контролни съобщения между обектите – фиг. 4.18.2
- времевата диаграма описва графика на състоянията от машината на състоянията - прилага се за RTприложения и системи – RTOS, ES (4.18.3)

Развоен, процесен и физически изглед

4.20

- Развойният изглед и асоциираният с него интерфейс изглед описват потребителските функции на приложението както и основните нефункционални изисквания
 - произтича от потребителското задание
 - в UML се специфицира с диаграма на потребителските случаи (4.15)
- Процесният изглед описва декомпозицията на разпределеното приложение с оглед на реализираните функции
 - представя основните блокове или компоненти
 - в UML се специфицира с клас-диаграма (статична), допълнена с една или повече динамични диаграми – най-често последователности или на дейностите (4.20.1)
- Физическият изглед описва цялата PCA на платформата + приложението – инсталация, конфигурация, разгръщане
 - компонентите са на ниво процесори или поне процеси
 - връзките между тях са на ниво комуникационни канали
 - представя нанасянето (или картирането – mapping) на компонентите от развойния изглед върху инфраструктурните възли (4.20.2)

ADL

4.22

- Architectural Description Language - графична спецификация на модели на разпределена со туерна архитектура
- свободно разпространявана среда за спецификация на ADL-модели AcmeStudio (<http://www.cs.cmu.edu/~acme/AcmeStudio/index.html>) с автоматична генерация на Java и C++

5.2

Съдържание

- ОО архитектури
 - абстракции, структури, отношения
 - анализ и принципи на проектиране
- Поточкови архитектури
 - пакетна обработка
 - архитектура с канали и филтри р и
 - контролна архитектура
- Контекстни архитектури
 - с хранилища
 - тип Черна дъска

Модел на изгледи

4.19

- 4+1 моделиране – представя PCA с 4 основни изгледа и един допълнителен – логически, развоен, процесен и физически + сценарий на приложние/функциониране, който често се придружава и от изглед на потребителските интерфейси – фиг. 4.19
- Сценарният изглед и асоциираният с него интерфейс изглед описват потребителските функции на приложението както и основните нефункционални изисквания
 - произтича от потребителското задание
 - в UML се специфицира с диаграма на потребителските случаи (4.15)
- Логическият изглед описва декомпозицията на разпределеното приложение с оглед на реализираните функции
 - представя основните блокове или компоненти
 - в UML се специфицира с клас-диаграма (статична), допълнена с една или повече динамични диаграми – най-често последователности

Потребителски интерфейс изглед

4.21

5. Обектни, потокови и контекстни модели на софтуерната архитектура

5.1

ОО принципи

5.3

- развитие на езиковите принципи при усложнена софтуерна архитектура – първоначално за симулационни модели (Simula67, C++); Интернет приложения (Java, C#); към компонентно базирани технологии
- прилагат се три основни принципа
 - капсулиране – видимост на функциите и прозрачност за имплементацията . Например скрит вътрешен контекст и процедури – частни променливи в класовете, неустойчиви; публичен интерфейс – устойчив
 - наследственост – адаптивност на кода чрез наследяване и допълване на спецификациите – т.е. от общо (родителски клас) към частно (наследен клас, дериват)
 - полиморфизъм – адптивна функционалност чрез развитие на наследяването
 - отмяна и предефиниране на атрибути в дериватите (вертикален полиморфизъм) или
 - презареждане на нов контекст за същия клас – хоризонтален полиморфизъм

ОО софтуерно инженерство

- Абстрактни типове данни
 - капсулиране на данните с функциите върху тях
 - стандартни библиотеки от типове
 - публични и частни атрибути на типовете
 - класовете са имплементации на АТД с публичен интерфейс от атрибути и операции
 - обектите са имплементации на класове, които се явяват техни «типове» – UML-спецификация на клас с +/- модификатори на достъпността на атрибутите и операциите – фиг. 5.4
 - Статични отношения между класовете:
 - конструкция на комплаксни класове от класове
 - композиция
 - наследяване
 - статична консистентност (т.е. логичност) на зависимите класове – като при базите данни
 - агрегация,
 - асоциация
 - Динамични отношения между класовете – обмяна на съобщения
- (N.B. ОО проектирането е ориентирано към мултикомпютърните архитектури)

Наследяване и полиморфизъм

- наследяването отразява взаимстване на повтарящите се атрибути – деривата наследява всички публични атрибути (без частните – възможни изключения)
- полиморфизмът е механизъм за диверсификация на дериватите при изпълнение – 5.6
- в UML наследяването се означава с триъгълна стрелка към основния клас
- в примера двата деривата се различават по методите на идентификация
 - клиентът зарежда cookie в браузъра си
 - регистрираният потребител изпраща парола и ползва отстъпка
 - и двете функционалности отсъстват в базовия клас

ОО анализ – диаграма на случаите

- анализът предхожда проектирането и имплементацията и се състои в структуриране на предметната област и представянето ѝ като набор класове с определена функционалност
- обикновено се състои в описание на потребителския сценарий чрез диаграма на случаите, от която се извлича и аналитичната (или принципна) клас-диаграма
- диаграма на случаите (Ivar Jacobson 1987) – пример за OPS (Order Processing System) 5.8:
 - определя типове потребители на системата – напр. клиент, счетоводство, доставка
 - определят се основните случаи, които ще се детайлизират като [една или повече] операции в етапа на проектирането – напр. случая добавяне на изделие в пазарската количка би изисквал и операция със складовата БД

ОО проектиране

- проектирането е самостоятелна фаза в развоината дейност на разпределените систем
 - може да се приложи подход, различен от този на фазата на анализа – потоков (event driven), контекстен (data driven), структурирен (с функции)
- целта му е декомпозиция на системата на технологични модули – при ОО – класове
 - класовете се описват с техния интерфейс т.е. публичните им атрибути и операции, и се специфицират след това на фазата на имплементацията
- различават се високо и ниско ниво на проектирането
- високото ниво идентифицира класовете напр. с приложение на CRC-карти и клас-диаграми за статичните отношения (specification/compile time) между класовете
- ниското ниво детайлизира проектираните класове и тяхното динамично взаимодействие (run time) с диаграми за взаимодействието (най-често с диаграми на последователността или на комуникациите) и на машината на състоянията (state machine) – като се използват диаграмите на случаите от фазата на анализ

Композиция, агрегация и асоциация

- **композицията** е дефинирана на клас като съставен от други класове
 - компонентите са активни докато и активен съставния клас и не се включват в други класове (пресилено ограничение за garbage collection – чрез структурите и деструктурите на класовете)
 - в UML – пълтен ромб към главния клас с етикети на мощността - 5.5.1
- **агрегацията** е аналогично отношение на класовете, но без изброените ограничения - 5.5.2
- **асоциацията** е обобщена композиция – 5.5.3; характеризира се с
 - име (етикет), което отразява свързващата функционална логика – напр. «Customer place» an/some Orders»
 - мощностите на асоцииране
 - 2 асоциативни типа на връзката между двата класа – задават тип композиция към инициращия клас
 - навигационната посока към инициращия клас – т.е. указателите на асоциираните класове са налични като атрибути в инициращия клас (пътна линия)
 - зависимост посока към зависимия клас – зависимия клас извиква операция на асоциирания клас или променя негов атрибут (пунктир)
 - инициращият клас може да асоциира повече от един класове

Обхват на наследяването и композицията

- и двете черти поддържат взаимстването на атрибути между класовете (reuse), но с различен обхват на приложение съгласно принципите:
 - наследяване се прилага при is-a отношение между деривата и базовия клас
 - композиция (или агрегация) се прилага когато отношението е has-a
 - пример: базови класове Person и University, класът Student може да бъде дериват на двата класа или да има атрибути с указатели към двата класа или комбинация от двата подхода
 - Student IS-A Person → Student е уместно да бъде наследствен дериват на Person
 - Student HAS-A University → Student е уместно да има атрибут с указател към University
 - наследяването е противоположно за капсулацията (локалността) на кода тъй като промяна на атрибут в базовия клас предизвиква каскадни промени в дериватите -
 - пример (5.7) - Student и Professor като деривати на Person (легитимно но ниска капсулация) и като агрегиращи PersonalHandler (с прозрачна конверсия на обръщението към атрибути)

ОО анализ – принципна клас-диаграма

- принципната КД е абстрактно описание на класовете на системата – по-близко до сценариите и функционалността, отколкото до имплементацията (не отчита производителност на модулите, технологии и технологичност на проектирането и експлоатацията)
- състои се от гранични, същностни и контролни класове (boundary, entity, control)
 - граничните класове се извличат от интерфейсите случаи и са ориентирани към имплементация
 - с GUI (Web форми, прозорци, браузър-плагини) или
 - като междинни интерфейси (middleware wrappers) към други системи
 - същностните класове отразяват информационния слой – напр. клиентската или пордуктова идентичност са същностни класове
 - контролните класове отразяват отделните случаи т.е. операциите, които свързват граничните и същностните класове
- пример – 5.9 – принципна КД на OPS

ОО проектиране – стъпка 1.

- прилагат се CRC карти (Class-Responsibility-Collaborator – Kent Beck & Ward Cunningham, 1989) и/или клас диаграми за пълно (а не принципно като при анализа) описание на класовете
- CRC картата на всеки клас таблица с описание на името, функционалните задължения (responsibility – заданията които изпълнява + контекста им) и списък клобориращи класове за изпълнение на тези задължения
- пример за OPS от 5.9: RegistrationPage и RegistrationController – 5.11

ОО проектиране – стъпка 2.

- описва се взаимодействието между обектите от ст. 1.
- прилагат се диаграми на последователността или на комуникациите
- моделът се състои от последователни стъпки, описани чрез обмен на съобщения
- пример – диаграма на последователността за случая Registration – описва обмена между класовете RegistrationPage и RegistrationController – 5.12:
 - в горната част на диаграмата са взаимодействащите обекти – с означения <object_name>:<class_name> (името на обект може да отсъства)
 - връзките отразяват дейностите на съответните обекти и носят съответните етикети – включително new за създаване на обект от клас-коллаборатор
 - в примера само обектите successPage и failurePage са именувани – за разлика от останалите класове – тъй като се предават алтернативно от RegistrationController към RegistrationPage

5.14

ОО проектиране – стъпка 4.

- ст. 4. е подробното описание на интерфейсите на всеки клас – избяват се атрибутите и операциите и тяхната публичност (с + и - в UML)
- публичната част от интерфейса е фиксирана и не трябва да де променя в следващата след проектирането фаза – имплементацията
- публичният интерфейс се състои главно от дефинирани константи и операции:
 - операциите в публичния интерфейс са 4 категории
 - конструктор
 - деструктор
 - аксесор
 - мутатор
 - определянето на публичните атрибути (константи) се базира на следните фактори
 - какви са външните стойности, които класът използва в своите операции – от CRC-диаграмата – напр. класът RegistrationPage използва Име и Парола (5.12)
 - какви са възможните състояния на класа от ДМС – те се включват като атрибути (но обикновено частни)
 - от мощността на асоциациите: 1..1 асоциация изисква скаларен атрибут-указател към асоциирания клас, а 1..* асоциация – атрибут-колекция (вектор)
 - други атрибути, необходими за изпълнение на операциите – обикновено са

5. Обектни, потребителски и контекстни модели

ФМИ/СУ * СИ * РСА

14

5.16

Потокови (Data Flow) архитектури

- [NB: тук в смисъл на софтуерни архитектури]
- представят обработката като последователност от трансформации (т.е. групи операции) върху последователност от набори структурирани еднотипни данни
- системата се декомпозира на функционални модули или подсистеми – паралелизъм по управление – аналогия с [нелинейните] конвейри
- интерфейсът между модулите може да е във формата на потоци (streams), файлове, канали (pipes, асинхронни потоци) и др.
- основният паралелизъм е по данни, тъй като ритъмът на обработка се задава от наличието на данни за обработка
- по тази причина – отсъствието или минимизирането и импликацията на контролния поток – ПА са подход и стил, приложим предимно при автоматизирани процеси на обслужване – напр. езикови компилатори, автоматизирани системи с пакетно обслужване като разпределените транзактивни системи, вградените системи

5.18

Пакетна обработка (Batch Sequential)

- най-старият модел на СА за обслужване в транзактивни системи и класическите ОС със стандартен файлов IO и редиректори
- приложението е скрипт с команди за изпълнение на съответните модули в UNIX, DOS, Tcl/Tk – напр.


```
myShell.sh
exec searching kwd <inpitFile >matchedFile
exec counting <matchedFile >countedFile
exec sorting <countedFile >reportFile
```

изходът се представят като методи и атрибути на класа

ОО проектиране – стъпка 3.

- ст. 3. описва динамичното поведение на по-сложните класове за целия им цикъл на живот – напр. контролните класове – с диаграми на машината на състоянието
- ДМС се извлича от диаграмите на случаите, в които участва дадения клас
- в ДМС отделните състояния означават стабилност на колекцията от поменливи на средата и от вътрешни променливи на класа
- вътрешните променливи на класа обикновено задават граничните стойности, с които се сравняват променливи на средата (условие за преход между състоянията на класа) и евентуално се изпълнява преход в друго състояние
- за по-сложните класове ДМС е съставна – включва и sub-state диаграми, но:
 - [правило] сложният клас е желателно да се представи от няколко класа ако логическата му функционалност не се описва от едно прокто изречение; това се отразява обратно и в CRC-модела

5.15

Обхват на ОО архитектурите

- предимства:
 - непосредствена връзка с потребителските сценарии и проблемната област
 - взаимстване (reuse) и капсулиране на имплементацията
 - лесно допълване чрез полиморфизма и класовете-derivati
 - устойчивост на системата поради защитеност на локалните атрибути
 - удобен преход към други модели и най-вече към компонентна архитектура
- възможни проблеми:
 - непредвидени странични ефекти при взаимодействието на много обекти, включително при асоциации 1..*
 - интерфейсите и вътрешната имплементация на класовете – макар и пордукт на отделни фази – не са толкова разграничени, колкото при компонентните архитектури; обикновено се разработват съвместно, което снижава нивото на абстракция (и сложност) на цялата архитектура, а също обичайно води до по-фина грануларност в сравнение с компонентните архитектури
 - наследствеността между класовете често води до грешки в спецификацията и следва да се прилага мн внимателно

5.17

Категории потокови архитектури

- топологията на пренос на данните между модулите се задава експлицитно с блок-диаграми (5.17)
- обработката е асинхронна
- модулите поддържат само интерфейс по данни, не и контролен интерфейс и не се адресират взаимно – адресацията е само чрез предаваните данни
- по механизма на свързване между модулите (т.е. на обмен) се разграничават
 - пакетна обработка (Batch Sequential)
 - филтрирани канали (Pipe & Filter)
 - контролни процеси (Process Control)

5.19

Приложимост на пакетната обработка

- данните (включително междинните резултати!) са оформени в пакети – файлове, т.е. с последователен достъп
- модулите се представят като програми, които се активират със скрипт или като резидентни модули, които сканират входните си файлове
- неприложима СА за интерактивен интерфейс
- широко приложение за асинхронни паралелни процеси – данните се декомпонират като множество входни файлове, а обработващите модули се репликират в множество възли – принцип на обслужване в пакетната фонова обработка – Condor, Boinc

Филтрирани канали (Pipe & Filter)

- приложението се декомпозира на източник на данните, филтри, канали (pipes) и консуматор на данните (sink)
- данните са последователни FIFO потоци (буфери, опашки) от байтове, символи или записи, които представят в последователен вид всички структури – вкл. и по-сложни, които се сериализират – в ОС marshalling/unmarshalling
- филтрите
 - трансформират потока данни – без необходимост да изчакват готовност на целия пакет за разлика от пакетната обработка!
 - записват изходните данни в канал, който ги предава на друг асинхронно работещ филтър
 - 2 типа филтри:
 - активен филтър – изпълнява операциите pull/push върху пасивни канали – каналите осигуряват съответните операции а инициативата е на филтъра. В Java PipedWriter и PipedReader класовете предоставят този интерфейс за канали
 - пасивен филтър – предоставя push/pull интерфейси на каналите
- каналите преместват – а по същество съхраняват – потока данни, които се обменят между два филтъра

Контролни СА

- прилагат се при вградените системи (BAC) – компютърно контролиране на процеси в реално време с или без човеко-машинен интерфейс
- при вградените системи управлението е на база на сканиране на порменливи на средата, извлечени като поток данни от сензори и управляващо въздействие чрез компютърно контролирани актуатори – напр. автомобилен ABS – 5.22
- при КСА процесът се разделя на няколко модула, но те са от 2 типа
 - контролни модули – за следене и манипулиране на променливите на средата и състоянието
 - изпълнителски модули – за управление на актуаторите
 - връзките между модулите са чрез поточни данни
- типове контролни потоци при КСА
 - контролирани променливи – характеристики на BAC (сила на ток, налягане и др. физически контроли на изпълнителните актуатори) – те се измерват текущо от сензорите и се съпоставят с контролните константи т.е. целевите стойности
 - входни променливи според проблемната област (скорост, налягане, температура, влажност, GPS координати)

Контекстни архитектури с хранилище

- макар и с управление по данни – за разлика от потоковите архитектури за пакетно обслужване на трансакции – тези архитектури поддържат интерактивните UI
- пример: клас-диаграма на университетска информациялна система – 5.24.1
 - класът Collector поддържа вектор на колекция от студентски записи и затова агрегира клас Student, като поддържа UI за извличане, добавяне и промяна на записите за студентите
 - класът Student е интерфейс към таблицата на студентите, чиито инстанции представят по един запис (т.е. ред) в нея
 - диаграмата на последователността 5.24.2 представя споделянето на данните чрез класа Student между няколко клиенти
- релационните СУБД са обичайната платформа за имплементация на тези архитектури, тъй като поддържат свързаност (консистентност) на разпределения достъп до данните, както и множество системни средства за операция, базирани на метаданни
- за по-висока отказоустойчивост и защита на данните се прилагат разпределени хранилища
- основен недостатък е статичната структура на данните – еволюция в структурата на релационните таблици се прилага трудно, струва скъпо и надеждността ѝ се проверява трудно

Диаграми на КАРх с черна дъска

- клас-диаграма на такава архитектура – 5.26.1
 - класовете-източници KnowledgeSource съхраняват специфичните правила за логически изводи, регистрират се в съответната ЧД, абонират се за оповестяване на промени в данните на ЧД и евентуално генерират реакции с изменения в локалния си или общ (ЧД) контекст; форматът на знанията и правилата за всеки ИЗ може да е специфичен
 - ЧД управлява общия контекст, регистрира промените в него, оповестява абонатите и регистрира евентуалните реакции, както и съхранява крайното решение
 - контролерът иницира ЧД, множеството на ИЗ, инспектира състоянието им и публикува крайното решение
- последователностна диаграма на архитектурата – 5.26.2
- блок-диаграма на КАЧД на система за туристически консултации – 5.26.3
 - обединява множество резервационни агенции – пътни, хотелски, за атракции, за коли под наем, кредитни и т.н.
 - клиентските заявки се публикуват на ЧД и се оповестяват съответните агенти, чрез реакциите на които се изготвят един или повече планове за туристическо пътуване и съответното финансиране
 - всички операции се иницират по данни, а се поддържа и UI: типично за КАЧД клиентският интерфейс през контролера е минимален – примерно еднократен, но интерфейсиът за управление на агентите може да е итеративен

Свързаност на филтрираните канали

- клас-диаграма на СА с филтри и канали 5.21.1 – активният модул е с пълтни интерфейсни линии
- филтърът е свързан с до 3 класа – източник на данните, консуматор и канал
- блокова и последователностна диаграма на ФКСА – 5.21.2
- ФКСА се организира лесно в пакетните ОС –
 - напр. в Unix `who | wc -l` означава пасивен канал между две операции – в случая `who` генерира списък от потребителите, `wc` брои думите в списъка (спрямо стандартни разделители); поддържат се канали с имена, а филтри могат да са произволни процеси в основен и фонов режим (fore- и background)
- макар, че управлението е по данни, паралелизма е управление и архитектурата е приложима когато обработката може да се раздели на асинхронни модули
- реализира модела производител/консуматор
- не се поддържа динамичен и интерактивен интерфейс – ограничение, което е предимство при дадени приложения
- приложението се ограничава от формата на данните в каналите – обикновено се използва ASCII код

Контекстни архитектури (Data Centric)

- характеризират се с централизирано хранилище на данните, които са достъпни за всички компоненти на системата, така че декомпозицията е на модул за управление на достъпа до данните и агенти, които извършват операции върху тях
- интерфейсиът между агентите и данните може да е явен – напр. RMI или RPC – или имплицитен – напр. транзактивен
- в чист вид КАРх не предвиждат преки комуникации между информационните агенти – 5.23
- модулът данни изпълнява операции по извличане или регистриране и промяна на записи – по 2 възможни модела:
 - хранилище (repository) – с активни (инициативни) агенти – хранилището е обикн. е организирано като СУБД, CORBA, UDDI или Web-услуги
 - черна дъска – с инициатива на модула данни – агентите са абонати за събития (event listeners), които настъпват при промяна в данните и на които абонатите отговарят реактивно – често при AI-разпределени приложения, охрантелни системи за разпознаване на звук и образ, системи за управление на бизнес ресурси – складове, транспорт

Контекстни архитектури с черна дъска

- ориентирани са главно към проблеми, решими с методите на AI – най-вече разпознаване на шаблони в различни области (първите приложения от края на 1970те са експертни системи в метеорология, изображения, звук, молекулярна химия)
- декомпозират решаването на проблеми също на два[+] дяла
 - черна дъска, съхраняваща данни – факти и хипотези т.е. еволюционни модели над фактите
 - източници на знания – паралелно работещи агенти, които съхраняват различни страни (данни, организирани като знания) от проблемната област – всеки ИЗ капсулира специфичен аспект от проблема и е отговорен за частни хипотези и решения като част от общото решение
 - [контролер – система за начало зареждане и управление на разпределеното приложение]
- запазва се блок-д-та от 5.23, но контролният поток е само от ЧД към ИЗ:
 - неяни (имплицитни) обръщения към регистрираните в ЧД агенти-източници
 - обръщанията възникват при промени в данните и се предават към абонатите за тези промени ИЗ, които изпълняват реактивно започените в тях логически правила за извод
 - този асиметричен механизъм на обмен е известен като модел `publish/subscribe` (pub/sub) в общите комуникации (вж. л-я 7.)
 - класифицират се като слабо-свързана (loosely coupled) PC поради асинхронния комуникационен модел с обмен на публикувани съобщения към абонатите (за разлика от силно свързаните (tightly coupled) системи с хранилища, където транзактивното обслужване е свързано със заключване на данните за конкурентен достъп – л-я 11.)

Обхват на КАРх с черна дъска

- подходяща архитектура за комплексни неизследвани и особено мултидисциплинарни проблеми които са
 - без детерминистично решение и с представяне на контекста във форматите на AI
 - неподходящи за търсене на решение с пълно обхождане на проблемния домен поради изчислителната сложност или непълнота/неточности в данните
- може да се генерират оптимално или няколко субоптимални решения или решения на частни подпроблеми
- за разпределена обратка с умерена скалируемост поради централизирания контекст
- проблем е еволюцията в структурата на контекста поради обвързаност с агентите на знания
- отсъствието на междуагентни комуникации води до необходимост от централизираната им синхронизация (например приоритетна) на достъпа до общия контекст
- трудно се формулира условие за край на обработката поради недетерминистичния характер на проблемите

6. Йерархични, асинхронни и интерактивни модели на софтуерната архитектура

Йерархични архитектури

6.3

- декомпонират системата по управление на йерархични модули - т.е. функциите се групират по йерархичен принцип на няколко нива
- координацията обикновено е между модули от различни нива (вертикална свързаност) и се базира на явни (т.е. "заявка-отговор") обръщения
- ниските нива функционират като услуги към непосредствените по-високи нива; услугите са имплементирани като функции и процедури или пакети от класове
- пълна прозрачност между нивата се постига при запазване на свързващите интерфейси, но имплементацията на услугите може да еволюира
- архитектурен модел на много ОС (Unix, MS .Net) и на протоколните стекове (TCP/IP); разслояване:
 - базови услуги – системните услуги се групират в модули за IO, транзакции, балансирано планиране на процеси, защита на информацията
 - междинен слой – "ядро" – поддържа проблемно-ориентирана логика – бизнес приложения, числова обработка, информационна обработка, като представя интерфейси към колекции от базовите услуги
 - потребителски интерфейсен слой – напр. команден екран, графични контролни прозорци, Shell скрипт интерпретатор

Диаграми на MS архитектура

6.5

- потоковата диаграма се използва за начално моделиране на изискванията към системата
 - потокова диаграма на OPS (Order Processing) – местата отразяват обработката, а дъгите – преноса на данните – 6.5.1
 - възел 1 – регистрация на заявките; в. 2 – валидиране и отказ (в. 4) или предаване на заявката; в. 3 приема или отказва заявка (в зависимост от изпълнимостта); в. 5 променя стоквата наличност и предава за фактуриране на в. 6; в. 7 обработва правилата за отказ и предава на в. 8 за уведомление (примерно друга оферта)
- при анализа се идентифицират
 - трансформиращите възли – променят формата на входните данни (напр. XML) към вътрешен формат – обикн. възлите с един вход и един изход
 - транзактивните възли – обработват входящите данни и ги насочват към един или друг изходен поток или пък нямат изходящи дъги
- от потоковата диаграма се извлича блокова диаграма на архитектурата – която е съставена от контролни и диспечерски модули (подпрограми) – съответстващи респективно на трансформиращите и транзактивните възли на потоковата диаграма – 6.5.2

Обхват на подпрограмните архитектури

6.7

- широко приложими разделяне на функциите по принципа отгоре-надолу
- приложими са и при OO имплементация
- проблем може да бъде достъпа до глобалните данни
- глобалните данни са модел на [разпределена] обща памет – затова са по-подходящи при мултипроцесорни машини – и обикновено аргументите на обръщението са указатели, а не стойности

- Йерархични архитектури
 - подпрограми и Master-Slave
 - слоевата архитектура и виртуални машини
- Асинхронни архитектури
 - буферирани и небуферирани модели
- Интерактивни архитектури
 - модел-изглед-контролер – I и II
 - представяне-абстракция-контрол

Йерархия с подпрограми (Main/Subroutine)^{6.4}

- традиционна архитектура, предхожда OO, базира се на процедури със споделен достъп до данните (само частична капсулация)
- декомпозицията е по управление, като комплексната функционалност на приложението се разделя на по-малки функционални групи – процедури и подпрограми – с цел тяхното споделяне между различни извикващи ги модули
- актуалните данни са параметри на обръщанията към изпълнителните функции и могат да се адресират по
 - указател – подпрограмата може променя техните стойности на същия адрес
 - стойност – подпрограмата получава стойностите като константи
 - име – подпрограмата използва като аргумент локалната стойност за съответното име (л-я 10.) – най-често това са локални имплементации на протоколи и други резидентни програми или динамични библиотеки
- главната програма управлява процеса на последователни обръщения към подпрограмите
- подпрограмите формират нефиксирана но ациклична слоевата йерархия – 6.4

Master/Slaves

6.6

- това е вариант на архитектурата с подпрограми, който е специализиран към поддържане на допълнителни нефункционални изисквания – най-вече
 - отказоустойчивост (fault tolerance) и надеждност
 - балансиране за ускорено изпълнение на заявките
- реализира се чрез репликиране на функционалните модули
- задача на M е алтернативно
 - да оцени адекватността на паралелно обработените резултати от S_n – съществуват протоколи за отказоустойчивост, идентифициращи грешните и верни резултати при ограничен брой на изпълнителните реплики
 - да извърши разпределяне на заявките прилагайки принципите за товарен баланс (л-я 3.)
- блок диаграма и клас диаграма на Master/Slaves архитектура – 6.6

Слоести архитектури

6.8

- групиране на различните нива в йерархията във функционално свързани слоеве от пакети класове, библиотеки от подпрограми (включени в т.нар. header files – заглавни файлове на проекта)
- интерфейсът на слоя се състои от интерфейсите на включените в него компоненти, а изпълняваната от тях функционалност – т.е. набора услуги – е протокола на слоя; интерфейсът му към нисколежащите слоеве се определя от техния интерфейс
- обработката се декомпозира на заявки от по-висок слой към непосредствения по-нисък слой
- възможно е прескачане ("bridge") в йерархията, но то е нетехнологично, тъй като изисква поддържането на пове4е от един интерфейс към слоеве с услуги; това се налага при необходимост от минимизиране на целия проект – напр. премахване на криптиращ слой
- протоколите на високите нива изпълняват приложно-ориентирани услуги, а на по-ниските – системно-ориентирани
- типично разслояване (6.8.1): потребителски интерфейс ↔ бизнес слой ↔ базови услуги ↔ услуги на ядрото
- клас диаграма на слоевата архитектура с имплементация на общ интерфейс от всички слоеве – 6.8.2

Компонентно-базирано разслояване 6.9

- основен подход за капсулирането на услугите в слой е формирането на компонент, който се описва със своя интерфейс – напр. .jar файл в JVM
- .jar файлът (създава се с `jar -cmf`) представя всички класове от по-ниските слоеве и включва класовете от слоя, който имплементира
- компонентите на отделните слоеве формират пакета на платформата – Java API
- всеки клас от .jar компонента е достъпен за приложенията чрез своя интерфейс – стига да е включен в променливата на средата `classpath`
- логическа организация на пакет от компоненти – пакетна диаграма 6.9

Виртуални машини 6.11

- виртуалните машини са слоест модел, който предоставя високо ниво на абстракция – програмен език или интерфейс за приложенията, при който скрива или обвива изпълнителната платформа
 - NB: VM представя основните абстрактни функции на системата като ги унифицира без да ги променя – напр. скрива интерфейсът към ОС – докато изпълнимите програми (C++) трябва статично да се прекомпилират за всяка ОС, както и за всеки тип процесор; понятието емуляция (с което неправилно виртуализацията се смесва) означава изпълнение на функциите на дадена система от друга система (с принципно различни функции или организация) – напр. емуляция на Unix върху MSDOS/Windows или емуляция на PDA и Smart/Mobile Phones от настолен компютър
- Unix VM – 6.11.1
- MS .Net VM – 6.10
- JVM – 6.11.3

Асинхронни архитектури 6.13

- базират се на неявни (implicit) асинхронни обръщения между обслужващите процеси
- асинхронният обмен може да бъде
 - в реално време (online) – без буферирание – и двата процеса трябва да са активни, но не блокират изчакващо в точката на обмен – 6.13
 - независим (offline) – с опосредяващ обмена процес-буфер на съобщенията; приемащият процес може да не е активен в момента на изпращане на съобщението и обратно
- активният процес генерира съобщения, а пасивните процеси ги получават и евентуално изпълняват реакция
 - прилагат SW-шаблоните Производител /Консуматор (Producer/Consumer) или Издател/Абонат ≡ Наблюдател (Publisher/Subscriber, Observer)
 - управлението е по събитие (event driven) – където събитието е издаване на съобщение от издателя и получаване на съобщение от абоната
- в независимия вариант процесът-буфер алтернативно може да служи като
 - централизатор Message Topic на всички издадени съобщения и да ги препраща тематично до абонатите – един-към-много обмен
 - резервирана опашка Message Queue за един-към-един обмен

Буферирани асинхронни СА 6.15

- системата е
 - контекстна (data-centric),
 - слабо свързана (не се чака потвърждение за получаването на съобщенията и обикновено не се получава отговор след обработката) – но с надежден обмен
 - декомпозира се на 3 части
 - генератори на съобщения (producers)
 - консуматори на съобщения
 - услуга за асинхронен буферизиран обмен на съобщения – MOM (Message Oriented Middleware)
- висока скалируемост, надеждност, p2p и CS приложения
- за системна поддръжка (мрежи, телекомуникации), бизнес приложения (булетини – новини, метеорология, групи по интереси; транзактивно банкиране и е-търговия)
- поддържа опашки (Message Queuing, MQ) и тематичен обмен (Message Topic, Publish/Subscribe Messaging P&S)
- атрибути на съобщенията са ID, заглавие (header) и тяло
- клиентите на системата обменят съобщения инициативно или пасивно, като адресацията е на базата на идентификатор, получен при началната регистрация на клиента в услугата за обмен

Модели на разслояване 6.10

- OSI: App ↔ Pre ↔ Ses ↔ Tra ↔ Net ↔ DLL ↔ Phy
- Web-услуги – 6.10 и л-я 8.: SOAP ↔ XML ↔ HTML ↔ TCP/IP
- Unix: shell ↔ core ↔ device drivers
- MS .Net: CLR ↔ JIT ↔ CTS
 - .Net е технология, която осигурява платформата CLR (Common Language Runtime) за изпълнение на приложения на C#, VB.NET, C++/CLI аналогично на JVM – 6.10
 - за прозрачност и преносимост приложенията се компилират до платформено-независим междинен език CIL (Common Intermediate Language),
 - по време на изпълнение CIL кодът (т. нар. „управляван код“) не се интерпретира като при други виртуални машини, а се компилира по начин, известен като JIT (Just In Time) компилация в платформено-зависим машинен код (native code) – за конкретната хардуерна платформа и операционна система
 - управлението на паметта, на нишките и процесите, защитата на паметта, верификацията и вътрешната компилация са системните услуги на CLR
 - CTS (Common Type System) дефинира всички базови типове данни и извършва конверсиите им. Тези типове са споделени между всички .NET езици и са стандартизирани в CLI.

Обхват на слоестите архитектури 6.12

- прилагат се за еволюционна развойна дейност, при която нивото на абстракция се повишава – принципа на проектиране е отдолу-нагоре, а не обратно
- всеки слой може да се разглежда като виртуална машина от определено ниво
- постига се
 - във високите слоеве – значителна прозрачност и преносимост на кода
 - в ниските слоеве – възможности за взаимстване на код (reuse) чрез промяна и добавяне на класове при запазен интерфейс на слоя
- подходяща за компонентни имплементации
- висок системен свръхтовар и по-ниска производителност – в сравнение с MS архитектурите
- свръхтоварът може да се преодолее с „мостове“ през слоевете, но това намалява предимствата и смисъла на обща виртуализация
- слоевете имат тенденция да скриват настъпването на изключения от по-ниско ниво

Небуферирани асинхронни СА 6.14

- системата се декомпозира на 2+ части
 - генератори на събития (sources)
 - слушатели на събития (event listeners)
 - регистратори на събития, които опосредяват обмена и по-конкретно поддържат асинхронността и неявното (непряко) оповестяване на слушателите
- архитектурен модел на SmallTalk приложенията:
 - л пасивни графични компонента-слушатели View_n се регистрират в активно (т.е. инициативно) пространство на събития EventSpace за съобщения от даден генератор на събития Model – 6.14.1
- клас диаграма на архитектурата – 6.14.2
 - класът Event Source осигурява операции за регистриране на слушател и за уведомяване за събитие
 - класът Event Listener осигурява операция за анализ на събитието и генериране на реакция
- подходящ модел за приложения с GUI и слабо-свързана логика, чиито модули се представят с машина на състоянията и имат недетерминистично поведение (поради което по-сложна настройка и тестване)
- налична е значителна поддръжка от междинни компоненти
- елемент на синхронност (👁) е началната регистрация
- сравнително ниска производителност и голям системен свръхтовар

MOM 6.16

- MS MQ
 - (http://en.wikipedia.org/wiki/Microsoft_Message_Queueing)
- IBM WebSphere MQ (бивш MQseries)
 - (http://en.wikipedia.org/wiki/WebSphere_MQ)
- JBossMQ (Java Message Server)
 - (<http://www.jboss.org/community/docs/DOC-10525>;
 - (http://java.sun.com/products/jms/tutorial/1_3_1-fcs/doc/jms_tutorialTOC.html)
- Oracle [бивш BEA] WebLogic JMS
 - (<http://e-docs.bea.com/wls/docs92/index.html>)
- (вж. и л-я 7.)

p2p (point-to-point) обмен

6.17

- обменът е 1:1 – всяко съобщение има точно 1 получател
- елементи: изпращач на съобщения, получател и асоциирана с получател опашка, която поддържа асинхронността на обмена
- съобщенията до даден клиент-консуматор се съхраняват в неговата опашка-буфер до извличането им или до изтичането на срока им
- пример – блокова диаграма на p2p обмен в EJB (Enterprise Java Beans – Java компонентна библиотека за бизнес приложения) – 6.17
 - получател е MDB (Message Driven Bean)
 - изпращач е клиентски процес
 - опашката може да се организира чрез JMS (Java Message Service <http://java.sun.com/products/jms/>) API – системно приложение за поддръжане на универсален асинхронен обмен

JMS комбинирана (p2p + P&S) СА

6.19

- клас-диаграма и д-ма на последователността – 6.19
- по отношение на услугата на обмена клиентите (производители и консуматори на съобщения)
 - се регистрират
 - откриват сесия за изпращане или приемане на съобщения
 - създават опашка или тема
- JMS (и др. MOM) поддържа следните контроли за надеждност и QoS на обмена
 - обмен с потвърждение от опашката/биюлетина
 - означаване на съобщението като обмен без загуба
 - установяване на приоритет на съобщенията
 - срок на съобщението (expiration)

Интерактивни софтуерни архитектури²¹

- поддържат интензивен потребителски интерфейс
- за целта декомпозицията на системата е на 3 функционални модула
 - модул за представяне (изглед) – с потребителски интерфейс – за представяне (в т.ч. графично или мултимедийно) на изходните данни и също намеса на потребителите в обработката (т.е. вход за данни и контрол)
 - модул данни – поддръжане на данните с базова функционалност върху тях
 - модул за управление – системни комуникации, управление на процесите, инициализиране и конфигуриране на модули данни, управление на изгледи
- поддържа множество (и то адаптивни) изгледи за даден набор данни
- слабо свързана архитектура, която поддържа явни и също неявни обръщения към метод – респ. RMI и модел регистрация уведомление (notification)
- две категории ИСА: PAC (Presentation-Abstraction-Control) и MVC (Model-View-Controller)
 - аналогията е P-V, A-M и C-C
 - прилагат различно управление:
 - PAC е с йерархично (разслоено) и разпределено управление, при което системата се формира от набор коопериращи агенти на три нива – базово ниво агенти на общи данни и бизнес логика, ниво на изгледите за локални данни и средно ниво агенти координатори на изгледите; всеки агент интегрира P, A и C компоненти
 - в MVC агентите са равнопоставени

MVC I

6.23

- компактна и базова двуделна имплементация на MVC, при която контролерът и изгледът са интегрирани в един модул C/V
- C/V се регистрира и присъединява към даден модел като се абонира за уведомления за промени в контекста, които представя в реално време, и служи като Вх/Изх на модула за данни – 6.23
- C/V
 - поддържа форми за потребителски вход – текстови полета, радиобутони (за алтернативен или множествен избор на опции) и др. виджети
 - при промяна във входните данни ги валидира и генерира заявка към модела с новото съдържание
 - представя резултата, който се генерира от модела (на базата на заложената в модела функционалност)
 - представя промените в контекста на модела, за които е абониран (без заявка от потребителя) – модел "активен контекст"
- MVC I е приложим за по-прости приложения с компактен GUI

Pub/Sub (P&S) обмен

6.18

- тази СА се базира на централизатор (hub), поддържащ асинхронния и непряк обмен на съобщения между издатели и абонати по теми (topics) – тип бюлетин
- инициативата в обмена принадлежи на източника на съобщението – на издателя спрямо бюлетина и на бюлетина спрямо абоната – така се постига максимална асинхронност
- вариант е устойчивия абонамент (durable subscription), при който абоната получава и съобщенията по дадена тема, издадени преди неговата регистрация в бюлетина
- блок-диаграма на P&S СА 6.18 – системата се базира също на JMS MDB/EJB, но за разлика от p2p при P&S крайните получатели на дадено съобщение могат да бъдат повече от един – всички регистрирани (и евентуално бъдещите!) абонати по темата (или темите), за които е издадено съобщението
- при разгърнатата P&S СА клиентите – издатели и абонати – са отдалечени разпределени процеси без никаква явна връзка помежду си, като абонатите обикновено изпълняват информационни услуги за трети клиенти – напр. сесии със СУБД

Обхват на асинхронните СА

6.20

- подходящи са за слабосвързани системи с устойчив неявен обмен на съобщения, при които обменящите процеси са анонимни не-заят идентичността на комплементарния процес/и (в т.ч. и неговия интерфейс!)
 - т.е. времева и локационна независимост
- висока скалируемост и заменимост на компонентите
- подходящ за динамично настройваеми разпределени изчисления (при асинхронен алгоритъм!)
- подходящи СА за пакетна обработка
- подходящи за интегриране на наследени приложения (legacy systems) в съвременни проекти
- независимостта между обменящите клиенти ограничава логиката на приложенията:
 - логиката на клиентите трябва да е независима от получаването (и неполучаването) на конкретни съобщения
 - не се идентифицира източника и няма пряк обмен с него
- усложнена логика на клиентите поради изискването за гъвкавост т.е. всеки клиент се самоконтролира (контраст с йерархичните и централизираните системи)
- възможност за тясно място (bottleneck) – по време (производителност на опашката/биюлетина) и по пространство (размер на опашката/биюлетина)

MVC

6.22

- основен модел за сърверни приложения с Web-клиенти за достъп – е-бизнес, е-управление, системи за потребителски профили и т.н.
- специализация: промени в контекста (данните) се представят динамично т.е. в реално време при отдалечени клиенти
 - изгледите се базират на интуитивни графични интерфейси с приложение на контекстно настройваеми "кожи" и фокусиране на интерфейса – етикети, бутони, изборни полета и др. компоненти от тип widget (в-ж л-я 9.)
- приложна компонентна платформа за проектиране на MVC е напр. Java Swing (<http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/package-summary.html>)
- трите дяла на MVC имат следната специализация:
 - контролерът регистрира подрежда и предава последователността от потребителски заявки; настройва изгледа вкл. динамично и управлява останалите модули на СА – стартиране, настройка, обмен
 - моделът изпълнява базовите функционални услуги, като капсулира контекста (непрозрачна обвивка на данните); при СА MVC I той не поддържа пряк интерфейс с присъединените към него изгледи
 - изгледът т.е. динамично настройваемо графично представяне на заявена част от контекста

MVC II

6.24

- при MVC II контролерът и изгледът са самостоятелни, а евентуално и отдалечени процеси
- допълнителна функция на контролера е да инициализира връзката между изгледа и модела и управлява обмена между тях
- контролерът и изгледът се регистрират в модела и се уведомяват разпределено за промените в контекста
- разделянето позволява самостоятелна имплементация и технологии за V и C
 - това способства за проектиране на сложна функционалност и също за самостоятелна еволюция на двата модула – по-специално на изгледите, които се поддържат от бързоразвиващите се графични технологии
- блок-диаграма, клас-диаграма и последователностна диаграма на MVC II – 6.24
 - инстанцииите на класовете V и C са "сдвоени", като множество двойки се поддържат от един модел
 - класът модел агрегира колекция от класове с различни функции върху базата данни

- блок-диаграма на MVC II CA, базирана на Java технологии – 6.25
 - JSP (Java Server Pages) се използва за V; EJB (Enterprise Java Beans) + JDBC (Java Data Base Connectivity) се прилагат за развитие на M; C може да се имплементира като поризволно сърверно приложение – напр. с Java Servlet технологията (сървелетите са Java сърверни приложения без потребителски интерфейс, които се инициализират от резидентни сърверни програми напр. Tomcat – подобно на аплетите, които обаче се изпълняват в клиентската част от брауъра)
 - контролерът получава потребителска заявка от графичен или текстов интерфейс (1), стартира необходимата инстанция на модела (2), селектира и стартира необходимия изглед (3) – с което управлението се предава към изгледа
 - изгледът получава данни от модела (4) и ги представя графично (5)
 - разгледайте аналогичните технологии в платформата MS .Net – ASP/ADO
- обхват на MVC
 - това е базовата архитектура за приложения с интензивен потребителски В/И с динамично представян на данните и с възможност за самостаятелна имплементация на модулите
 - поддържа се от множество професионални платформи за шаблонно развитие на приложенията
 - не поддържа агентно-базиран информационен обмен, характерен за системите с редуциран потребителски интерфейс – автономни и вградени системи, роботи, автономни агенти и др.

РАС-приложение

6.27

- примерно РАС-приложение (клас- и последователностна диаграма 6.27) за преглед на отдалечен странициран документ
 - с 4 бутона – за първа, предишна, следваща и последна страница – поддържани от агентите $A_2 \div A_5$ съответно
 - A_6 – за графична интерпретация на страниците от документа по съответен стандарт
 - A_1 е агента за достъп до документа в БД
 - C_1 приема заявките от C_i ($i = 2 \div 5$), настройва A_1 на съответната страница, приема я от него и я предава на заявителя – A_1 няма нужда от P_1
 - C_1 съобщава на C_i за настройки на бутоните от P_i (напр. избледняване на бутони "следваща стр." и "последна стр." ако прегледа достигне последната страница) и предава на C_6 съдържанието, което се представя от P_6
 - A_i поддържат контекста на съответните агенти – напр. предпочитан изглед на бутон, текущото му състояние
 - A_6 поддържа контекста на представяната страница – напр. декодиращ метод, кеширани страници

7.1

7. Паралелни алгоритми за графи и търсене

Графи – дефиниции

7.3

- ◆ графът $G(V, E)$ е двойка крайни множества на върховете (vertexes) и дъгите (edges); дъгата $e \in E$ е двойката $e=(u, v)$ ($u, v \in V$), или наредената двойка $e=(u, v)$ – насочен граф
- ◆ съседни (adjacent) са върховете, свързани с дъги (и в двата типа графи); N.B. $\exists(u, v) \in E \Rightarrow u$ е съседен на v , не и обратното
- ◆ път в графа е последователност (наредено множество) от върхове $P=\{v_1, v_2, \dots, v_k\}; \forall(v_i, v_{i+1}) \in E, 1 \leq i \leq k$
- цикъл в граф: път $P=\{v_1, v_2, \dots, v_k\}; v_1 \equiv v_k$; ациклични графи; прост цикъл
- ◆ пълен граф: всички върхове са съседни
- ◆ свързан граф: съществува път между всяка двойка върхове
- ◆ подграф: $G'(V', E') \subseteq G(V, E); V' \subseteq V$ и $E' \subseteq E$; свързан подграф (connected subgraph) – подграф на свързан граф, който запазва свойството свързаност между подмножеството върхове
- ◆ свързан компонент: свързан подграф $G^*(V, E' \subseteq E)$ на ненасочения свързан граф $G(V, E)$, за който $|E'| = \min$ (7.3)

- РАС е развитие на MVC, което поддържа агентен обмен на съобщения
- системата се състои от множество специализирани (т.е. с различни функции) агенти, декомпозирани на трите модула – P, A и C;
- декомпозицията на даден агент разделя неговия потребителски интерфейс (P) от функционалността, която поддържа (A) и от модула му за обмен с др. агенти (C) – 6.26
 - презентационния модул на агента е опция (съществуват агенти-посредници без потребителски интерфейс)
 - контролният модул е задължителен, освен комуникациите с отдалечени агенти, той управлява достъпа до функциите на агента – P и A са слабосвързани процеси без пряк обмен
 - абстрактният модул капсулира данните и операциите на агента

Обхват на РАС

6.28

- прилагат се за интерактивни системи от коопериращи специализирани информационни агенти
- слабосвързана разпределена система – комуникациите са неблокиращи асинхронни
- добри възможности за заменимост, еволюция на агентите, ескалиране на системата
- поддържа еднакво многопроцесни и многопроцесни разпределени приложения
- значителен свръхтовар особено при групови комуникации
- непряк (бавен) обмен между контекста и представянето му
- имплементацията на A и P е зависима от тази на C – затруднение при проектирането
- усложнени операции за откриване на броя и идентифициране на текущите агенти

7.2

Съдържание

- Представяне и проблеми при графите
- Алгоритми за обхождане и път – последователни и паралелни версии
- Търсене в ширина, в дълбочина – последователни и паралелни версии
- Оптимизиране с α - β minimax търсене

Свойства в графите

7.4

- дърво: свързан ацикличен граф; с-ва:
 - $|E| = |V| - 1$
 - единствен път между всяка двойка върхове
- маркиран граф $G(V, E, W); W(E) \rightarrow \mathfrak{R}$; тегло на граф и тегло на път (суми)
- двуделен граф: $\exists V_1$ и $V_2: V_1 \cap V_2 = \emptyset$ и $V_1 \cup V_2 = V$ и $\forall(u, v) \in E \Leftrightarrow (u \in V_k; v \in V - V_k)$ (възможна е бисекция)

Представяне на графите

- за алгоритмични цели графът $G(V, E)$ с $|V| = n$ се представя с матрици или списъци
- матрични форми:
 - матрица на съседство (adjacency matrix): $A_{n \times n}(a_{ij})$: $a_{ij} = \{1 \Leftrightarrow (v_i, v_j) \in E \mid 0\}$; за ненасочените графи A е симетрична (7.5.1)
 - тегловна матрица (weight matrix) за маркираните графи $W_{n \times n}(w_{ij})$: $w_{ij} = \{0 \Leftrightarrow i = j \mid w(i, j) \Leftrightarrow (v_i, v_j) \in E \mid \infty\}$;
 - матрица на свързност ($\{\text{connectivity} \mid \text{reflexive} \mid \text{transitive closure}\}$ matrix) $C_{n \times n}(c_{ij})$: $c_{ij} = \{1 \Leftrightarrow \text{съществува ацикличен път } P_{ij} \mid 0\}$;
- списъчна форма: G се представя с n линейни свързани списъка със съседите на всеки връх – 7.5.2

Обхождане

- прилагат се 3 базови алгоритъма, които имат паралелни версии:
 - търсене – “в дълбочина” (depth-first) или “в широчина” (breadth-first): избира се произволен възел v и всички негови съседни се маркират с v ; в следващите стъпки итеративно се избира произволен немаркиран възел w и се повтаря първата стъпка за маркиране на съседите; сложност $O(n + m)$ където $|V| = n$ и $|E| = m$
 - с използване на матрицата на свързване
 - с разделяне на подграфи

...Обхождане чрез матрицата на съседство

- всеки ацикличен път между два произволни върха на G е не по-дълъг от броя върхове $n \Rightarrow C = B^{n-1}$; на практика итеративните изчисления са $B, B^2, B^4, \dots, B^m, C$, където $m = (n-1)/2$ алгоритъмът има $\lceil \lg(n-1) \rceil$ итерации от матрични умножения
- когато размера на графа не е по степените на 2, m е най-малката степен на 2, по-голяма от $(n-1)$ – напр. за $A_{7,7}$ $C = B^8$
- следващата стъпка е получаване на матрицата на свързване компоненти $D_{n \times n}$ от C , където $\forall d_{jk} \in D = \{v_k \text{ (ако } c_{jk}=1) \mid 0\}$ ($0 \leq j, k \leq n-1$) – ред j на D съдържа върховете, към които v_k образува свързан компонент с индекс x , където x минималния индекс на ненулев d_{jk}
- този метод е удобен за паралелна обработка тъй като се свежда до матрични изчисления с паралелизъм по данни; реда на изчисление е $A \rightarrow B \rightarrow C \rightarrow D$ в четири последователни цикъла за паралелни итерации (последния е за намиране на индекса на компонентите x)
- сложността на умножението на логически матрици е $\lg(n-1)$ итерации – всяка с оценка $O(\lg n)$, което дава обща сложност на алгоритъма $O(\lg^2 n)$

...Обхождане чрез разделяне

- сложността на стъпка 1 за търсене на покриващото дърво в матрицата на съседство A_i (n/p) $\times n$ на подграфа G_i е $O(n^2/p)$
- сложността на стъпка 2 за сливане на покриващите дървета по двойки се състои от $\lg p$ сливания с $O(n)$ сложност на всяко от тях, така че общата сложност на тази стъпка е $O(n \lg p)$
- общата сложност на паралелната версия на алгоритъма е $O((n^2/p) + (n \lg p))$

Проблеми върху графи (с паралелно решение)

- обхождане
- минимално покриващо дърво (minimum spanning tree)
- най-къс път
- откриване на циклите
- задачата за търговския пътник

Обхождане чрез матрицата на съседство (transitive closure)

- методът построява матрицата на свързване на графа $C_{n \times n}$, като степенува логически матрицата на съседство $A_{n \times n}$
- по принцип логическото умножение на булеви матрици (каквито са $A_{n \times n}$ и $C_{n \times n}$) представлява операцията (с логическо умножение и събиране!)

$$A_{n \times n} \times B_{n \times n} = C_{n \times n} : c_{ij} = (a_{i1}b_{1j}) + (a_{i2}b_{2j}) + \dots + (a_{in}b_{nj})$$
- първата стъпка на метода е построяване на спомагателна матрица $B_{n \times n}$ която се получава от $A_{n \times n}$ с разполагане на 1 по главния диагонал; тогава $\forall b_{jk} \in B = \{1 \text{ (ако има ацикличен път с дължина } 0 \text{ или } 1 \text{ от } v_j \text{ до } v_k) \mid 0\}$
- следващите стъпки са итеративно намиране на продукта B^m ($m < n$), елементите на който отразяват съществуването на ацикличен път между съответните върхове с дължина не по-голяма от m ; – пример 7.8

Обхождане чрез разделяне (adjacency matrix partitioning)

- методът се състои в разделяне на матрицата на съседство $A_{n \times n}$ по редове на p части – колкото са обработващите процесори – като процесор P_i обработва подграфа $G_i(V, E_i)$, който се състои от съответните върхове и дъги – 7.10
- обработката на съответния подграф е откриване на неговото покриващо дърво чрез търсене, след което покриващите дървета на подграфите се сливат по двойки
- сливането на две покриващи дървета S_1 и S_2 – които имат най-много $(n-1)$ общи върхове – се извършва като за всяка дъга $(u, v) \in S_1$ се проверява дали върховете u и v присъстват в S_2 – ако да – S_1 и S_2 се сливат в тези върхове, в противен случай се минава към следващата дъга на S_1
- алгоритъмът се състои главно в стъпка 1: локално търсене на покриващите дървета на подграфите и след това стъпка 2: сливане по двойки

Път в маркиран граф

- маркиран граф $G(V, E, W)$; $W(E) \rightarrow \mathbb{R}$; тегло на път $W(p) = W(v_1, v_2, \dots, v_k) = \sum_{i=1, k-1} W(v_i, v_{i+1})$; най-къс път
- проблеми:
 - най-къс път за двойка върхове
 - най-къс път за направление $d \in E$ от останалите върхове
 - най-къс път с начало $s \in E$ до останалите върхове
 - най-къс път между всички двойки върхове – матрица D с най-късите пътища – свойство: най-късият път между двойка върхове съдържа най-късите пътища между вложените двойки върхове (“Optimality principle”)
- методи за построяване на D :
 - маркиращ алгоритъм на Dijkstra (greedy метод)
 - алгоритъм на Floyd (динамичен метод)

Маркиращ алгоритъм на Dijkstra

- базира се на временна и крайна двойна маркировка на върховете j според пътищата им до тях от дадено начало s :
 - етикет $d(j)$ = дължината на най-късия път (s, j), минаващ само през върхове с крайна маркировка
 - етикет $p(j)$ = предшестващия j връх в (s, j)
- алгоритъм:
 - стъпка 1: крайно маркиране на s :
 - $d(s) = 0; p(s) = \emptyset$
 - временно маркиране на останалите върхове j : $d(j) = \infty; p(j) = \emptyset$
 - стъпка 2: ако k е последния връх с крайна маркировка, достижимите от него върхове j с временна маркировка се маркират:
 - $d(j) = \min\{d(j), d(k)+d_{kj}\}$
 - $p(j) = \{k; (d(j) = d(k)+d_{kj}) \mid p(j)\}$
 - стъпка 3: маркировката на върха с най-малко $d(j)$ става крайна; ако има върхове с временна маркировка \Rightarrow стъпка 2;
 - край

Алгоритъм на Floyd за най-къс път между всички върхове

- базира се на
 - Optimality principle: $k \in P_{ij}(k, i, j \in V)$, ако P_{ij} е най-късия път, тогава и $P_{ik} P_{kj}$ са съответните най-късии пътища и
 - “триъгълната операция” $w_{ij}(k) = \min\{w_{ij}, w_{ik} + w_{kj}\} \forall i, j \neq k$
- за “триъгълната операция” се доказва, че ако се приложи върху всички стойности на тегловната матрица на графа $k=1, 2, \dots, n$, то всички стойности на получената матрица са равни на най-късите пътища
- алгоритъмът стартира с модификация на тегловната матрица $W^{(0)}$, в която

$$w_{ij}^{(0)} = \min\{w_{ij}, (i, j) \in E \mid \infty, (i, j) \notin E \mid 0, i = j\}$$

Последователна версия на алгоритъма на Floyd

- състои се от n матрични итерации, всяка с n^2 проверки, така че сложността е кубична - т.е. същия резултата както n пъти изпълнение на алгоритъма на Dijkstra; на практика обаче тази програма е с по-бързо изпълнение на отделните цикли и като цяло с по-кратък код
- псевдокод:


```
array D[n, n], W[n, n]
for i, j = (1, 2, ..., n) D[i, j] ← W[i, j]
for k = (1, 2, ..., n)
  for i = (1, 2, ..., n)
    for j = (1, 2, ..., n)
      D[i, j] ← min{D[i, j], (D[i, k] + D[k, j])}
return D
```

Паралелна версия на алгоритъма на Floyd

- версия за изпълнение от $p = n^2$ процесора, [логически] свързани в двумерна мрежа като процесор P_{ij} изчислява последователно $w_{ij}^{(k)}$ (от стойностите на $w_{ij}^{(k-1)}, w_{ik}^{(k-1)}$ и $w_{kj}^{(k-1)}$) за $0 < k \leq n-1$
- броят на последователните итерации е $\lceil \lg n \rceil$
- псевдокод (междинните резултати T е необходимо да се получат предварително, за да се избегне конфликтно четене и запис с последния израз):


```
array D[n, n], W[n, n], T[n, n]
forall i, j = (1, 2, ..., n) in parallel do D[i, j] ← W[i, n]
repeat  $\lceil \lg n \rceil$  times
  forall i, j, x = (1, 2, ..., n) in parallel do
    T[i, x, j] ← D[i, x] + D[x, j]
  forall i, j = (1, 2, ..., n) in parallel do
    D[i, j] ← min{D[i, j], T[i, 1, j], T[i, 2, j], ..., T[i, n, j]}
return D
```

Паралелна версия на маркиращ алгоритъм на Dijkstra

- $V_p = \{s\}$
- forall $v \in (V \setminus V_p)$ do
 - if (s, v) exists
 - then $d(v) = w$
 - else $d(v) = \infty$
- endfor
- while ($V_p \neq V$) do
 - select vertex u : $d(u) = \min\{d(v) \mid v \in (V \setminus V_p)\}$
 - $V_p = V_p \cup \{u\}$
 - forall $v \in (V \setminus V_p)$ do
 - $d(v) = \min\{d(v), d(u) + w_{uv}\}$
- endwhile
- $O(n^2)$ - и $O(n^3)$ ако се търсят най-късите за всички двойки възли

...Алгоритъм на Floyd за най-къс път между всички върхове

- в първата (от n) стъпка $w_{ij} = w_{ij}^{(1)}$ - т.е. триъгълна операция спрямо връх 1 (ако пътя през връх 1 е по-къс от дъгата, той я заменя) - резултат $W^{(1)}$
- при втората стъпка триъгълната операция се прилага спрямо $W^{(1)}$ и възел 2: $w_{ij}^{(2)} = \min\{w_{ij}^{(1)}, w_{i2}^{(1)} + w_{2j}^{(1)}\}$ (т.е. най-къс път само през върхове 1 и 2 - ако съществува такъв) - резултат $W^{(2)}$
- рекурентно: $w_{ij}^{(k)} = \min\{w_{ij}, k=0 \mid \min\{w_{ij}^{(k-1)}, w_{ik}^{(k-1)} + w_{kj}^{(k-1)}\}, 0 < k \leq n-1\}$
- най-късите пътища между всички върхове се намират след n -тата стъпка с резултат $W^{(n)}$
- на практика търсената $D = W^{(m)}$ където $m = \lceil \lg(n-1) \rceil$ - т.е. за $n=7$ $m=3$ като матриците по степените на 2 се получават чрез стандартно умножение

Паралелна версия на алгоритъма на Floyd

- версия за изпълнение от $p = n^2$ процесора, [логически] свързани в двумерна мрежа като процесор P_{ij} изчислява последователно $w_{ij}^{(k)}$ (от стойностите на $w_{ij}^{(k-1)}, w_{ik}^{(k-1)}$ и $w_{kj}^{(k-1)}$) за $0 < k \leq n-1$
- броят на последователните итерации е $\lceil \lg n \rceil$
- псевдокод (междинните резултати T е необходимо да се получат предварително, за да се избегне конфликтно четене и запис с последния израз):


```
array D[n, n], W[n, n], T[n, n]
forall i, j = (1, 2, ..., n) in parallel do D[i, j] ← W[i, n]
repeat  $\lceil \lg n \rceil$  times
  forall i, j, x = (1, 2, ..., n) in parallel do
    T[i, x, j] ← D[i, x] + D[x, j]
  forall i, j = (1, 2, ..., n) in parallel do
    D[i, j] ← min{D[i, j], T[i, 1, j], T[i, 2, j], ..., T[i, n, j]}
return D
```

...Паралелна версия на алгоритъма на Floyd

- на k -тата итерация алгоритъмът проверява за нов път от i до j , минаващ през повече от 2^{k-1} върха и през по-малко от 2^k върха, който да е “по-къс” от текущия най-къс път, и ако такъв връх е x , дължината се записва в T
- в последния израз разделянето на контекста по процесори, избягва конфликта между новите и старите стойности в D

Задачи за търсене

- задачите за търсене са много широк клас и произтичат от разнообразни приложни области – най-често с представяне на проблемната област в термини от теорията на графите – и само сравнително неголяма част от тези алгоритми могат да се обработят на последователни архитектури за приемливо време
- търсенето се осъществява в различни структури данни – в зависимост от приложната област – поради което съществен е метода на търсене, който пряко произтича от обработваната структура (най-често в графи и дървета)
- основни методи:
 - търсене с разделяне (divide and conquer)
 - търсене в дълбочина (depth-first search, DFS)
 - търсене в ширина (breadth-first search, BFS)
 - оптимално (хибридно, евристично) търсене (best-first search)
 - търсене с разклоняване (branch and bound, BB)
 - оптимизиране – търсене на оптимуми (alpha-beta minimax search)

Паралелни версии на търсене с разделяне в сортиран списък

- при мултикомпютрите се изгражда логическо дърво от процеси/процесори, което съответства на дървото за търсене; напр. върху процесорен хиперкуб е необходима предварителна фаза на картиране (mapping) така че да се минимизират междупроцесните комуникации (по дължина респ. време)
- при мултипроцесорите наличието на обща памет улеснява достъпа към областите на подпроблемите

Търсене в дълбочина

- по същество това са алгоритми за обхождане на графи (подобно на алгоритмите за покриващо дърво) – проверява се дървовидна структура за дадена стойност на атрибут на някой/и от върховете (и евентуално неговата позиция)
- за цела графите се представят със списък на съседство
- специфично за търсенето в дълбочина е, че обхождането на списъците продължава до намиране на връх, чиито съседи (елементите от неговия списък на съседство) са били вече проверени; след това с връщане назад на минимално разстояние обхождането продължава в нова посока (неизследвания връх)
- пример: ако сме стартирали от връх v и сме регистрирали дъгата (v, w) където w е непосетен, в следващата стъпка стартираме рекурсивно с w
- обратно – ако в горния сценарий w е вече посетен, не се преминава към следваща рекурсия и v остава връх, по чиито дъги се търси друг непосетен връх – т.е. именно проверката на върховете е в нарастваща дълбочина на дървото
- за удобство се приема конвенцията търсенето в наследниците (съседите) на даден връх да става отляво надясно; всеки проверен връх получава етикет-индекс DFI с реда му в последователността от проверки (в рамките на графа – не само в съответния клон)

Паралелно търсене в дълбочина

- DFS алгоритъмът е последователен по природа, за паралелно търсене е необходимо да се изследва матрицата на съседство (вместо списъка на съседство) и се въвежда списък на немаркираните съседи – $U(v)$ – подмножество на списъка на съседите на v , ако връх w бъде посетен и маркиран, той се изключва от $U(v)$ (както и от всички списъци на немаркирани съседи, в които присъства)
- резултатът е във формата на два списъка – на дъгите и на клоните – като списъка дъги $ARC_LIST(v)$ е всъщност списък на съседните върхове на v , а

Търсене с разделяне в сортиран списък

- последователно търсене с двоично разделяне - псевдокод:


```
location(index low, index high, x)
{
  middle = [(low + high)/2]
  if (x == Elements[middle]) return middle
  else if (x < Elements[middle]) return location(low, middle - 1, x)
  else return location(middle + 1, high, x)
}
```
- последователно търсене с многократно разделяне: всяка подобласт се решава рекурсивно в общата последователност - псевдокод:


```
Procedure Divide&Conquer(Input, Output)
  Divide(Input, Input1, Input2, ..., InputM)
  for i = 1, M do
    Divide&Conquer(Input, Output, i)
  endfor
  Combine(Output1, ..., OutputM, Output)
End Divide&Conquer
```

Пример за паралелно търсене с разделяне

- n мерен вектор от сортирани елементи $S = \{E_1, E_2, \dots, E_n\}$ се претърсва за стойност x от p -процесорна архитектура с обща памет ($p < n$)
- S се разделя на подвектори и всеки процесор P_i обработва последователните елементи $\{E_{n(i-1)/p+1}, E_{n(i-1)/p+2}, \dots, E_{in/p}\}$ като прочита x в CR режим и ако за P_k $E_{n(k-1)/p+1} \leq x \leq E_{kn/p}$ – тогава се търси локално $j: x = E_{n(k-1)/p+j}$, резултатът е $output = (k-1)n/p + j$
- псевдокод:


```
Procedure Parallel_Divide&Conquer(Input, Output)
  Divide(Input, Input1, Input2, ..., Input_p)
  for i = 1, p do in parallel
    Parallel_Divide&Conquer(Input, Output, i)
  endfor
  Combine(Output1, ..., Output_p, Output)
End Parallel_Divide&Conquer
```

Търсене в дълбочина – процедура

- параметър на процедурата е свързан граф, зададен със списък на съседство и стартов връх v , а резултат – индексите на върховете му (стартирайки с $v = 1$) - 7.26
- псевдокод:


```
Procedure DepthFirst(A)
  mark every vertex unvisited
  i = 1
  DepthFirstSearch(v)
Procedure DepthFirstSearch(v)
  mark v i
  for (each w adjacent to v) do
    if (w unvisited) DepthFirstSearch(w)
  i++
  endif
  endfor
End DepthFirstSearch
End DepthFirst
```
- сложност на последователния алгоритъм: n индексирания и m проверки $O(n + m)$ – мощностите на V и E

Търсене в ширина

- търсене в ширина стартира от начален връх (корен) и проверява всички върхове на разстояние една дъга от него, след това – на две дъги и т.н. до проверка на всички върхове – на практика се построява минималното покриващо дърво (в немаркиран граф/дърво пътя/клона се измерва в брой дъги)
- отново конвенцията за проверка е отляво надясно и всеки проверен връх получава етикет-индекс BFI с реда му в последователността от проверки (в рамките на графа – не само в съответното ниво-разстояние от корена) – резултатът е дърво, маркирано с индексите BFI (7.28)
- процедурата се базира на образуване на опашка от проверените съседи на текущия корен, върховете в която след това стават корени за търсенето на следващото ниво:

Търсене в ширина – процедура

7.29

```
Procedure BreadthFirstSearch
mark every vertex unvisited
initialize Queue with start vertex v
i = 1 /* BFI
while (Queue not empty) do
remove the top q of Queue /* v in the beginning
for each vertex w adjacent to q
if (w unvisited) then
mark w = i
i++
place w in the bottom of Queue
endif endfor endwhile
End
```

- сложността на последователния BFS е като на DFS (но с по-добри възможности за паралелна имплементация): n индексирания и m проверки $O(n + m)$ - мощностите на V и E
- прилагат се два подхода за паралелно BFS:
 - търсене по върхове (vertex-by-vertex BFS, VPBFS)
 - търсене по нива (level-by-level BFS, LPBFS)

Оптимизиране

7.31

- това е клас от задачи за търсене на оптимум на дискретни функции напр. при търсене на най-добър ход в игра (с противник)
- прилага се метода на α - β minimax търсене с изчерпателна проверка на всички възможни ходове (представени с върхове в дърво) и използване на връщане-назад; метода е от класа на ограниченото търсене в дълбочина с функция-критерий, която оценява възможните наследници на текущия връх
- най-добрата от оценките на наследниците се присвоява на родителя, избира се съответния наследник и респективно се оценяват неговите наследници (максимизиране) – 7.31.1
- търсенето в сценария на игра (неизвестен ход на противника) трябва да отчети не само оптималния ход (т.е. достижимо състояние) на фазата максимизиране, но и оценките на следващите достижими състояния, поради което при α - β minimax към родителя се предават също “най-малко лошите” оценки на различните наследници от всяка негова дъга (фаза минимизиране) – (7.31.2)

α - β е в р и с т критерий

7.33

- при α - β minimax оптимизирането се прилага евристича функция-критерий (α - β pruning) за ограничаване на търсенето
- стойността α е долната граница на оценката, която може да получи върха без да бъде отхвърлен; респ. β е горната граница (която може и да не бъде максималната оценка)
- правилата за контрол на търсенето чрез тези евристични параметри са:
 - търсенето не продължава след връх, за който играча (maximizer) получава α -оценка, не по-малка от β -оценката на противника (minimizer)
 - търсенето не продължава след връх, за който противника получава β -оценка, по-голяма от α -оценката на играча
- с помощта на такава функция се прочита дървото на достижимите състояния от решения, които не могат да бъдат оптимални (но които съгласно “чистата” minimax стратегия подлежат на изследване) – т.е. на фазата максимизиране се премахват от разглеждане ходове, за които се установи че оценката ще е под текущия праг и на фаза минимизиране – при оценки над прага (пример 7.33)

8.1

8. Паралелни алгоритми за матрици, изрази и сортиране

Паралелно търсене в ширина – вариант VPBFS

7.30 Procedure ParallelBreadthFirstSearch_VP(ALM, EM, U)

```
mark all vertices “unvisited”
v ← start vertex
mark v “visited”
instruct processor(i) where 1 ≤ i ≤ k /* k-node system
for j = 1 to k do
if (k*(j-1)+1) ≤ EM(v)
delete v from U(ALM(v, k*(j-1)+1))
endif endfor endinstruct
initialize Queue with v
while (Queue not empty) do
extract v from Queue
for each w ∈ U(v) do
mark w visited
instruct processor(i) where 1 ≤ i ≤ k
for j = 1 to k do
if (k*(j-1)+1) ≤ EM(v) then
delete w from U(ALM(w, k*(j-1)+1))
endif endfor endinstruct
add w to Queue
endif endwhile
End
```

- отново се използва списъци на немаркираните съседи $U(i)$ ($1 \leq i \leq n$), матричен списък на съседството $ALM_{n \times (n-1)}$ и вектор на валентността (end-marker vector) $EM_{1 \times n}$

α - β minimax оптимизиращата стратегия

7.32

- α - β minimax оптимизиращата стратегия е рекурсия със следните атрибути:
 - генератор на ходове – функция, която връща списък на достижимите състояния за всеки играч
 - играч – може да бъде в позиция maximizer («играч») или minimizer («противник»)
 - функция-критерий – стойностите й се наричат статични оценки
 - критерий за край – индикатор за край (пределна дълбочина) на рекурсията – при който се избира (по функция-критерий) ортималния ход при върха-родител; такива критерии обикновено са пределна дълбочина в брой нива или «игрално време»

Паралелно α - β minimax оптимизиране

7.34

- този клас паралелни алгоритми обикновено има ниски стойности на насищане на ускорението (т.е. ниско ниво на паралелизма)
- паралелната обработка се базира на следните подходи:
 - паралелна генерация на ходовете и изчисляване на статичните оценки
 - паралелно търсене (обхождане)
- паралелната генерация и статични оценки има сравнително пониска линейност от паралелното търсене
- при паралелното търсене се разделя дървото T на клонове – BFS подход; в някои версии за по-големи системи се допуска и разделяне на заданието по нива при което процесорите се организират в логическо дърво

8.2

Съдържание

- Паралелна обработка за
 - префиксни изчисления
 - матрици
 - изрази
 - сортиране

Префиксни изчисления

8.3

- дефинират се върху наредено множество от реални компоненти $\{x_1, x_2, \dots, x_n\}$ и асоциативна бинарна операция върху тях \otimes – напр. събиране и умножение на реални числа, минимум и максимум на две числа, конкатенация на низове и логическите операции върху два булеви операнда; за простота някъде \otimes е записана като сума («+») по-надолу, но операцията не е непременно комутативна!
- префиксните изчисления (или суми) са стойностите (8.3.1)
 $S_1 = x_1,$
 $S_2 = S_1 + x_2,$
...
 $S_n = S_{n-1} + x_n$
- префиксните изчисления се прилагат при изчисление на полиноми, CAD системи, диспечеризация, сливане на списъци, обработка на графи и др. поради което са микрокодирани като операции в някои специализирани процесори
- префиксните суми се изчисляват паралелно в двоично дърво с дълбочина $d = (\lceil \lg n \rceil + 1)$ за $d+1$ стъпки към корена и още d стъпки към листата – общо $2d+1$ стъпки – 8.3.2
ФМИ/СУ * КН * СПО

3

Матрични изчисления

8.5

- произтичат от всички проблеми, които се решават със средствата на линейната алгебра и някои типични задачи за обработка на матрици се приемат като еталонни алгоритми за изследване на производителността на паралелните системи
- матриците са пълни или плътни (dense) – респ. без или с малък брой нулеви елементи или разреждени (sparse) – с малък брой ненулеви елементи
- паралелните алгоритми се дават във вид за плътни квадратни матрици ($n \times n$), останалите случаи могат да бъдат обобщени
- основния подход за паралелна матрична обработка е декомпозицията – блокова или решетъчна (“checkerboard partitioning”)

8.7

Решетъчна декомпозиция

- с оглед на базовата операция матрично или матрично-векторно умножение, решетъчната декомпозиция може да се изпълни по различни начини в зависимост от съотношението на размера на матриците и векторите (респ. $n \times n$ и $n \times 1$) и броя обработващи процесори:
 - $p > n^3$ - всички умножения на елементите на операндите могат да се извършат едновременно като n копия на матрицата се разполагат последователно на съседни възли и колоната-вектор се репликира също n пъти
 - $p \geq n^2$ - при матрично-векторно умножение всички произведения могат да се извършат паралелно, при матрично-матрично умножение се прилагат следните подходи
 - всеки стълб (или ред) от резултата се обработва паралелно на една стъпка - общо n последователни стъпки
 - за n последователни стъпки се получават събираемите, от които е съставен всеки компонент на резултата
 - $p < n^2$ - матриците се обработват по части като се прилага блокова декомпозиция - по редове за първия операнд и по стълбове - за втория

8.9

Матрично-векторно умножение с

блокова декомпозиция

- при n -процесорна архитектура директния подход е всеки възел да зареди ред от матрицата и колоната на вектора и да изчисли съответния елемент на резултата-вектор:
Procedure ParallelMatrixVector_Row(A,B,C)
begin
for $i = 0$ to $n-1$ do in parallel
C[i] = C[i] + A[i,0:n-1]*B[0:n-1]
endfor end
- при което оценката за всеки процесор и за системата е $O(n)$ (за n процесора резултата е n^2 операции); очевидно най-удобно в този случай е приложението на блокова декомпозиция по редове
- ако $p < n$ се прилага горната схема на блокова декомпозиция по редове (циклична или групова - 8.9); по-добро балансиране (т.е. ефективност) се постига при кратност на отношението $n:p$
- ако $p > n$ и $n:p = k$ за ефективна обработка е необходимо всяка група от k процесора да си разпредели съответен ред от матрицата и част от векторната колона (което е всъщност решетъчна декомпозиция)

Префиксно изчисление в хиперкуб

8.4

- за изчисление на префиксните суми на n елемента в n -процесорен хиперкуб всяка двойка съседни елементи x_i и x_{i-1} се разполага в съответния процесор p_i и в края на обработката в него се получава S_i
- за целта (8.4):
 - стъпка 1: в p_i $S_i \leftarrow x_{i-1} \otimes x_i$ ($i = 1, \dots, n$; “+” представя \otimes)
 - стъпка 2: $k = 2$; p_i чете $S_{i,k}$ от p_{i-k} и $S_i \leftarrow S_{i,k} \otimes S_i$ ($i = k+1, \dots, n$)
 - стъпка 3: $k = k + k$; преход към стъпка 2 (докато $k < n$)
- псевдокод:
ParallelPrefixComput(X, \otimes , S)
step0: /*initialization
forall $i = 1..n$ do in parallel
p[i] reads X[i]; S[i] = X[i]
endfor
step1: forall $i = 1..n$ do in parallel
S[i] = S[i-1] \otimes S[i]
endfor
step2: $k = 2$
while ($k < n$)
forall $i = k..n$ do in parallel
S[i] = S[i-k] \otimes S[i]
endfor
 $k = k + k$
endwhile
end
ФМИ/СУ * КН * СПО

ртиране

4

Блокова и решетъчна декомпозиция

8.6

- блоковата декомпозиция се извършва или само по колони или само по редове; при плътните матрици всеки блок съдържа еднакъв брой колони респ. редове
- броят на блоковете p е желателно да бъде кратен на броя процесори r за по-добро балансиране, като обикновено $p = k * r$ (за скалируемост на алгоритъма)
- разпределението на блоковете по процесори може да бъде групово или циклично – 8.6
- решетъчната декомпозиция се извършва едновременно по колони и по редове с еднаква честота в двете направления

8.8

Матрично-векторно умножение

- $C_{1 \times n} = A_{n \times n} * B_{1 \times n} \Leftrightarrow c_i = \sum_{j=0, n-1} a_{ij} b_j$, където $C_{1 \times n} = [c_0, c_1, \dots, c_{n-1}]^T$, $A_{n \times n} = [a_{ij}]$ и $B_{1 \times n} = [b_0, b_1, \dots, b_{n-1}]^T$
- оценката на последователния алгоритъм е квадратична (ако се приеме умножението на два елемента и добавянето им към текущия векторен елемент като базова операция):
Procedure MatrixVector(A, B, C)
begin
for $i = 0$ to $n-1$ do
begin
C[i] = 0
for $j = 0$ to $n-1$ do
C[i] = C[i] + A[i,j]*B[j] /* basic operation
endfor
endfor
end
- паралелното матрично-векторно умножение може да се изпълни както с блокова, така и с решетъчна декомпозиция

Матрично-векторно умножение с

8.10

решетъчна декомпозиция

- при горните условия за контекста n^2 -процесорна архитектура директния подход е да се формира процесорна решетка като всеки възел зареди съответен елемент от матрицата а колоната на вектора се зарежда в първия ред от n процесори
- паралелния алгоритъм се изпълнява в 3 стъпки:
 - стъпка 1: разпространяване на вектора във всички редове на процесорната решетка
 - стъпка 2: локално умножение на двойката елементи на матрицата и вектора
 - стъпка 3: сумиране на елементите на резултата по редове
- по конвенция резултата се разполага в диагоналните процесорни елементи от решетката; в хиперкуб със същия брой процесори обработката е по-бърза поради по-високата валентност на възлите (респ. по-бързото разпространение на междинните резултати)
- този алгоритъм може да се приложи както в MIMD, така и в SIMD архитектури

Матрично-матрично умножение

- $C_{n \times n} = A_{n \times n} * B_{n \times n} \Leftrightarrow c_{ij} = \sum_{k=0, n-1} a_{ik} b_{kj}$, където $C_{n \times n} = [c_{ij}]$, $A_{n \times n} = [a_{ij}]$ и $B_{n \times n} = [b_{ij}]$
- т.е. c_{ij} е продукт от A_i и B_j (съответно ред и колона)
- при умножение на повече от две матрици се използва последователно асоциативността на операцията (която не е комутативна - т.е. зададения ред не може да се нарушава): $C = C_1 C_2 \dots C_n = ((C_1 C_2) C_3) \dots C_n$
- псевдокод:


```

Procedure MatrixMatrix(A, B, C)
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      C[i,j] = 0
      for k = 0 to n-1 do
        C[i,j] = C[i,j] + A[i,k]*B[k,j]
      endfor
    endfor
  endfor
end
      
```

Матрично-матрично умножение в двумерна процесорна решетка – код, 8.13

8.13

```

Procedure ... Step2
  ParallelMatrixMatrix_2d(A, B, C) for Pij where i,j = 0 to n-1 do in parallel
Step1 C[i, j] = A[i, j] * B[i, j]
for k = 0 to n-1 do endfor
  for Pij where i,j = 0 to n-1 do in parallel Step3
    if i>k then C[i, j] = A[i, j] * B[i, j]
      A[i-1, j] ← A[i, j] endif
    if j>k then B[i, j-1] ← B[i, j] endif
  endfor
Step2 endfor
end
      
```

... Матрично-матрично умножение в тримерна процесорна решетка

8.15

- стъпка 1: елементите a_{ij} и b_{ij} се зареждат в процесор $p_{(ni+j)}$ (процесори $0 \div n^2-1$)
- стъпка 2: разпространение на операндите до останалите процесори
- стъпка 3: в процесор p_{ijk} $c_{ijk} = a_{ji} * b_{ik}$ (след зареждане на необходимите един или два операнда от съседите)
- стъпка 4: след сумиране $c_{ij} = \sum_{k=0, n-1} c_{ijk}$ резултата c_{ij} се намира в процесор $p_{(ni+j)}$
- пример за $n = 2$ (8.15)

Решаване на система линейни уравнения

8.17

- идеята на различните методи за решаване на СЛУ е привеждане на разширената матрицата $Ab_{n \times n+1}$ към горно-триъгълна форма (съответстваща на редуциране на променливите) чрез елементарни операции по редове така че под главния диагонал остават само нулеви елементи, а диагоналните елементи са единици - фаза елиминирание (forward elimination); след това се извършва фазата заместване (back-substitution), при която A се трансформира в единична матрица, а в стълба b се съдържа решението на системата

Матрично-матрично умножение в двумерна процесорна решетка

- алгоритъмът се обработва от в $n \times n$ затворена процесорна решетка (SIMD) и стартира със зареждане на елементите на операндите a_{ij} и b_{ij} в процесора p_{ij} - при това само n процесора (по главния диагонал) съдържат двойка елементи за умножение
- паралелния алгоритъм се изпълнява в 3 стъпки:
 - стъпка 1: за комбиниране на подходящите двойки елементи n пъти се извършва ротационно местене на B -елементите нагоре и на A -елементите надолу
 - стъпка 2: локално умножение на двойката елементи на матриците
 - стъпка 3: разпространение на локалните междинни резултати към съседните възли надолу и нагоре за n итерации - след което резултата c_{ij} се съдържа в p_{ij}

Матрично-матрично умножение в тримерна процесорна решетка

8.14

- алгоритъмът се обработва от в $n \times n \times n$ процесорна решетка (SIMD) или хиперкуб като $\exists q: n = 2^q$ т.е. $p = 2^{3q}$
- при горното условие $n = 2, 4, 8 \dots$ и номерацията на процесорите има формата $p_{ijk} = p_x$ за $x = in^2 + jn + k$ ($i, j, k = 0, 1, \dots, n-1$; $x = 0, 1, \dots, n^3-1$), което съответства на номерацията в хиперкуб (8.14)
- ако архитектурата с n^3 процесори е решетка, а не хиперкуб (т.е. валентността на възлите е константа 4, а не $\text{lb}(n^3) = 3q$), този алгоритъм е в сила, но преноса на операнди няма да бъде само между съседни възли
- алгоритъмът изпълнява паралелно n^3 умножения с което обработва n^2 елементи от резултата:

Система линейни уравнения

8.16

- системата линейни уравнения $\sum_{j=1, n} a_{ij} x_j = b_i$ има матричното представяне $A_{n \times n} x_{1 \times n} = b_{1 \times n}$ с решение $x = A^{-1} b$ - при условие, че A е неизродена матрица (т.е. редовете и стълбовете ѝ не са линейно зависими);
- [диagonalна е матрица с ненулеви елементи само по главния диагонал; когато ненулевите елементи на диагоналната матрица са само единици, тя е **единична** матрица I_n като $AA^{-1} = A^{-1}A = I_n$; **тридиagonalна** е матрица с ненулеви елементи само по главния и двата съседни диагонала (т.е. за които $|i-j| \leq 1$); **долно-триъгълна** е матрица с ненулеви елементи само над главния диагонал и обратно - **горно-триъгълна** е матрица с ненулеви елементи само под главния диагонал]
- при представяне на коефициентите (т.е. елементите на матриците) с плаваща запетая решението $A^{-1} b$ често поражда числова нестабилност; на практика се прилага метода на LU-декомпозицията (в математиката - метод на Гаус), който числово е по-стабилен и се обработва около три пъти по-бързо

LU-декомпозиция

8.18

- това е приложен метод за генериране на $n \times n$ матриците L и U , за които е в сила:
 - L е единична долно-триъгълна
 - U е горно-триъгълна
- фазата елиминирание започва с добавяне на подходящи изрази към всички уравнения с изключение на първото, с което се елиминира първата променлива; по същия начин се елиминират последователно променливите от следващите уравнения - в общия случай за да се елиминира i -тата променлива от j -тото уравнение най-напред се умножава i -тото уравнение с a_{ji}/a_{ii} и полученото еквивалентно уравнение се изважда от j -тото уравнение
- определя се базов ред (уравнение), който се използва за нулиране на елементите под главния диагонал в колона i , (pivot)

- сложността на този последователен алгоритъм в двете фази е с кубична $O(n^3)$, тъй като:
 - елиминирание – за всеки ред i цикълът по k се изпълнява $(n - i + 2)$ пъти, а вложеният цикъл по $j - (n - i)$ пъти т.е. общо за фазата $\sum_{i=1, n} (n - i + 2)(n - i) -$ което дава линейна оценка
 - заместването е с квадратична оценка

```

Procedure GaussianElimination(A)
begin
    /* Forward elimination
    for i = 0 to n do
        begin
            max = i
            for j = i + 1 to n do
                if(abs(A[j, i]) > abs(A[max, i])) then max = j
            for k = i + 1 to n do
                t = A[i, k]; A[i, k] = A[max, k]; A[max, k] = t
            for j = i + 1 to n do
                for k = n + 1 to n do
                    A[j, k] = A[j, k] - A[i, k]*A[j, i]/A[i, i]
                end
            end
        end
    begin
        /* back-substitution
        for j = n to 1 do
            t = 0
            for k = j + 1 to n do
                t = t + A[j, k]*X[k]
            X[j] = (A[j, n+1] - t)/A[j, j]
        end
    end
    
```

Балансиране на LU-декомпозицията

- броят на обработващите процесори намалява с изпълнението на стъпките, което води до по-ниска ефективност на обработката; този ефект се наблюдава при повечето методи за решаване на СЛУ
- по принцип разделянето на проблема по редове (уравнения) е свързано с определяне на базовия ред и предаване на неговите параметри до останелите процесори, след което всеки процесор обработва един (или повече) от редовете
- аналогична обработка може да се извърши и с разделяне по колони, но в повечето случаи обработката по редове постига по-добро бързодействие

Паралелна обработка на аритметични изрази

- паралелизъмът при решаването на изрази се задава с дървото на разбора, а при потоковите архитектури - с графа на зависимостите
- еквивалентните изрази могат да имат различни дървета на представяне и съответно различна степен на паралелизъм - напр. E_5 и E_6 (от 1.) са подходящи съответно за последователна и паралелна обработка и имат различно представяне с дърво на разбора и различен паралелизъм - 8.23.1
- за съпоставка на последователната и паралелната обработка на един израз (респ. на еквивалентни изрази) се използват следните критерии (8.23.2):
 - брой стъпки (последователни или паралелни операции) за решаване на изрза
 - брой процесори - т.е. ниво на паралелизма (при MIMD архитектура)
 - общ брой операции

Сортиране

- сортирането на структури от елементи по техен ключ (стойност) е задача, която се изпълнява от компилатори, редактори, системни модули за управление на паметта и процесите и много приложения
- сортирането се разграничава на вътрешно и външно, като при второто само част от сортираните елементи се разполагат в основната памет, а останалите са във външната памет
- сортирането се базира на разделяне на елементите по групи и сравняването им в рамките на групата (в крайна сметка сравняването им по двойки), така че има две последователни фази - разделяне и сливане
- фазата разделяне присъства винаги тъй като операцията сравнение-размяна е бинарна, следователно последователността за сортиране трябва да се раздели по двойки операнди; допълнително предимството на разделянето е, че обработката по части позволява паралелизъм

Паралелни версии на LU-декомпозицията

- паралелната обработка на двете фази от LU декомпозицията може да се извърши с разпределяне на алгоритъма между процесорите по редове или колони на матрицата $A_{n \times (n+1)}$
- пример - когато размера на системата е от порядъка на n , фазата заместване при паралелна обработка по колони може да се извърши в следните $(n - 1)$ стъпки:
 - стъпка 1 - вход долно-триъгълна матрица A и вектор B - напр. 8.20 за $n = 4 - (n - 1)$ процесора обработват паралелно изразите от вида $b_i^{(1)} = b_i + a_{i1}x_1, i = 2, \dots, n$ и $x_1 = b_1$
 - стъпка 2 - $(n - 2)$ процесора обработват паралелно изразите от вида $b_i^{(2)} = b_i^{(1)} + a_{i2}x_2, i = 3, \dots, n$ и x_1, x_2 са известни
 - стъпка $k - (n - k)$ процесора обработват паралелно изразите от вида $b_i^{(k)} = b_i^{(k-1)} + a_{ik}x_k, i = k + 1, \dots, n$ и x_1, x_2, \dots, x_k са известни

Аритметични изрази

- състоят се от оператори и операнди със скоби за явно задаване на реда на операциите
- при синтактичния разбор (parsing) изразите се преобразуват в дърво; когато операциите в тях са бинарни - и дървото е бинарно - 8.22
- изчислението на израза става рекурсивно по неговото дърво, което логически е еквивалентно на задача за обхождане на бинарно дърво
- елементарни изрази са тези, в които всеки операнд (променлива) участва само веднъж
- достатъчни условия E да е елементарен израз са:
 - $E = x_i$; x_i е променлива
 - $E = G \otimes H$; G е прост израз и $\otimes \in \{+, -, \}$
 - $E = G \otimes H$; G, H са прости без общи операнди и $\otimes \in \{+, -, *, /$
- пример: $E = x_1 * x_2 * x_3$ е елементарен, но $H = x_1^2$ не е
- еквивалентни изрази са тези, които приемат еднакъв набор аргументи (по брой и тип) и връщат един и същ резултат за всеки набор от стойности на тези аргументи - пример: $E_5 = (x_1x_2 + x_3)x_4 + x_5$; $E_6 = x_1x_2x_4 + x_3x_4 + x_5$

Аритметични изрази в SIMD и MIMD

- допълнителна характеристика за израза е възможността за обработката от SIMD архитектура, при която паралелно може да се изпълнява само една елементарна операция (от всички процесорни елементи) - пример: еквивалентните изрази
 - $E_3 = (((x_1 * x_2) + x_3) * x_4) + x_5$; x_7 ;
 - $E_1 = (((x_1 * x_2) * (x_4 * x_5)) + (x_3 * (x_4 * x_5))) + ((x_5 * x_6) + x_7)$ и
 - $E_2 = (((x_1 * x_2) + x_3) * (x_4 * x_5)) + ((x_5 * x_6) + x_7)$,
 са с различен паралелизъм и сложност (брой стъпки) по отношение на SISD, SIMD и MIMD архитектура - (8.24)
- паралелната обработка на даден израз по принцип се извършва като за всеки връх на дървото се планира процесор:


```

repeat
    for each vertex x do in parallel
        if (children(x) known) then
            compute x
            remove children from the tree
        endif
    endfor
until only root left
            
```
- времевата сложността на такъв алгоритъм е $O(\ln b)$ където \ln е броя операции и процесори, а стойността на обработката е $O(\ln b \ln)$
- поради зависимостта по данни броя процесори може да се намали без да се увеличи времето за решаване на израза - средностатистически $p_{opt} = n / \ln b$; в този случай стойността е $O(n)$

Методи за сортиране

- два основни метода се прилагат от различни сортиращи алгоритми:
 - сливане - последователността за сортиране се разделя на две равни по размер части, които на свой ред се сортират рекурсивно; след това двете сортирани части се сливат - при този метод сравняването («избора) на елементите става във втората фаза - сливането («easy split / hard join»)
 - разделяне - последователността за сортиране се разделя на две равни по размер части като всеки елемент от първата е по-малък от кой да е елемент от втората; процеса на разделяне продължава рекурсивно за тези части до изчерпване, след което подредените елементи се сливат в сортирана последователност - при този метод сравняването (и избора) на елементите става в първата фаза - разделянето («hard split / easy join»)
- ефективността и бързодействието на паралелните версии на сортиращите алгоритми зависят значително от използваната архитектура

Представяне на сортирането с мрежи

- сортирането се представя с мрежи или графи, от които лесно се извлича топологията и процесите на паралелното сортиране
- сортиращите мрежи са комбинация от компаратори - логически устройства, които извършват операцията сравнение-размяна (8.27)
- формално компараторът е четириполюсник - устройство с два входа $I_{1,2}$ и два изхода $O_{1,2}$ - които имат следните свойства:
 - $O_1 = \min(I_1, I_2)$
 - $O_2 = \max(I_1, I_2)$
- компаратори, които нямат общи входове-изходи, могат да функционират паралелно
- компараторите се свързват в компараторни каскади (comparator stages), като изход[ите] на един компаратор е/са вход[ове] на следващ - функционирането на компараторите в каскада е последователно

Представяне на сортирането с графи

- сортирането може да се представи и с графи, които обикновено са бинарни дървета, конструирани по следния начин (8.29.1):
 - листата на дървото са входове, в които се разполага несортираната последователност
 - вътрешните върхове изпълняват операцията сравнение-размяна върху последователностите, които се съдържат в техните наследници
- тъй като сортирането има две фази - разделяне и сливане - които се изпълняват последователно в този ред, понякога то се представя като две свързани бинарни дървета (за двете фази - 8.29.2) - или еквивалентно - като едно дърво, на което дъгите са двупосочни и фазата разделяне съответства на движение към листата, а фазата сливане - на движение към корена
- от сортиращото дърво лесно се извлича паралелизма на обработката: върховете от едно ниво могат да се изпълняват паралелно от различни процесори (което съответства на обработка в дълбочина)

Сортираща мрежа odd-even

- сортиращата мрежа на базовия алгоритъм за n входни елементи се състои от n компараторни каскади; всяка каскада се състои от $(n-1)$ паралелно работещите компаратори $[i: i+1]$ - съответно за четните и за нечетните елементи (8.31)
- броят на компараторите е $n(n-1)/2$
- предимството на сортирането «четни-нечетни» (освен простотата) е запазване на принципа за локалност на операциите сравняване-размяна и също скалируемостта на алгоритъма и балансиране на операциите между процесорите в рамките на всяка итерация, но ефективността от приложението на n^2 компаратори е ниска
- в паралелна система отделните итерации могат да се изпълняват последователно от n процесора
- Упр: да се сметне ефективността и паралелизма

Пример: Odd-Even merge

- пример: $A = \{2, 6, 10, 15\}$ и $B = \{3, 4, 5, 8\}$ са двете равни по размер сортирани последователности за сливане в S (сортираща мрежа: 8.33):
 - стъпка 1: $A_{\text{odd}} = \{2, 10\}$, $B_{\text{odd}} = \{3, 5\}$, $A_{\text{even}} = \{6, 15\}$, $B_{\text{even}} = \{4, 8\}$;
 - стъпка 2: $S_{\text{odd}} = \{2, 3, 5, 10\}$, $S_{\text{even}} = \{4, 6, 8, 15\}$;
 - стъпка 3: $S_{\text{odd-even}} = \{2, 4, 3, 6, 5, 8, 10, 15\}$;
 - стъпка 4: $S_{\text{odd-even}} = \{2, 3, 4, 5, 6, 8, 10, 15\}$;
- дълбочината на рекурсия е логаритмична; при всяка итерация операциите сливане и сравняване-размяна се изпълняват $n/2$ пъти, върху n процесора те се изпълняват за 1 стъпка - следователно времето за обработка остава логаритмично, а стойността на обработката е $O(n \log n)$ - която е оптимална в сравнение със всеки последователен сортиращ алгоритъм

Сортиращи мрежи

- формално сортиращите мрежи са насочени графи със следните свойства (8.28.1):
 - върховете, от които има само излизаща дъга (ребро) са входове, а тези към които има само влизаща дъга са изходи на графа
 - останалите върхове (освен входовете и изходите) са компаратори
 - изходите на сортиращата мрежа съдържат сортирана последователност от стойностите, които се записват във входовете - т.е. сортиращите мрежи се състоят от компараторни каскади
- с помощта на сортиращи мрежи алгоритмите за сортиране могат да се проектират систематично с формални средства
- пример (8.28.2): мрежата на сортирането «мехурче» се състои от $n(n-1)/2$ компаратори в $(2n-3)$ каскади, паралелизмът на алгоритъма е $(n-1)$ и при наличие на такъв брой процесори алгоритъмът ще се изпълни за $(2n-3)$ стъпки
- сливаща мрежа е такава сортираща мрежа, на която входовете се делят на две [наредени] равномошни множества и ако на всяко от двете множества входове се подаде сортирана последователност, на изходите се получава общата сортирана последователност

Сортиране «четни-нечетни»

- на практика се прилагат предимно алгоритми от класа сортиране чрез сливане; при паралелната обработка най-често се реализира сортиране «четни-нечетни» и битонично сортиране (които са от класа чрез сливане) - причина за което е не само добрите оценки на тези алгоритми, но и това, че схемите (последователностите) на сравнения при тях не зависят от контекста
- базовата версия на сортирането «четни-нечетни» (Odd-even transposition sort) извършва операцията сравнение-размяна последователно в двуфазни итерации:
- в първата фаза се сравняват и разменят само четните елементи и техните [нечетни] съседи със следващ по-голям индекс т.е. $[j: i+1]$
- във втората фаза на итерацията сравненията са по алтернативните нечетни индекси

Сортиране «четни-нечетни» чрез сливане

- този алгоритъм (Odd-even merge sort) изпълнява сортиращо сливане на две сортирани последователности с еднакъв размер и рекурсивно разделяне на по-късите последователности по четни и нечетни индекси
- псевдокод:


```
Algorithm OddEven(A, B, S) /* A,B sorted subsequences of S
begin
if A, B are of length 1 then Compare-Exchange-Merge
else
begin
form A_odd, B_odd, A_even, B_even /* step 1
compute in parallel
OddEven(A_odd, B_odd, S_odd) /* step 2
OddEven(A_e..x, B_e..x, S_e..x)
S_odd-even = Merge(S_odd, S_even) /* step 3
S_odd-even = OddEvenInterchange(S_odd-even) /* step 4
end
endif end
```

Паралелен Odd-Even merge

```
Procedure OddEvenMerge(L[1:n])
Model: n-processor PRAM
Input: L[1:n]; n=2**k, L[1:n/2] and L[n/2+1:n] sorted
Output: L[1:n] sorted
if (n=2) then
if (L[1]>L[2]) then exchange(L[1], L[2]) endif
else
OddEvenSplit(L[1:n], Odd[1:n/2], Even[1:n/2]) /*separate list elements
/*of odd and even indices
/*recursive sorting
OddEvenMerge(Odd[1:n/2])
OddEvenMerge(Even[1:n/2])
for i = 1 to n/2 do in parallel
L[2i-1] = odd[i]
L[2i] = even[i]
end in parallel
for i = 1 to n/2 do in parallel
if (L[2i]>L[2i+1]) then exchange(L[2i], L[2i+1]) endif
end in parallel
endif
end OddEvenMerge
```

9. Потоково, функционално и SIMD програмиране

Съдържание

- Потоково програмиране – особености и езици
 - VAL
- Функционално програмиране
 - SISAL
- Програмиране в SIMD-архитектури
 - C*
 - FORTRAN90

Императивно и потоково (data flow) програмиране

9.3

- с конвенционалните императивни езици се създават програми, в които:
 - реда на изпълнение на отделните операции и команди се задава от програмиста (изпълнението на програмата е като прочитането на книга)
 - променливите могат да променят стойността си многократно и да се използват за различни (евентуално еднакви по тип) резултати
 - зависимостите по данни не се задава явно и откриването ѝ не е тривиално особено ако се използват команди за преход от типа на goto или ако променливите се използват за съхраняване на пазлични междинни резултати
- с езиците за потоково програмиране се създават програми за потоковите архитектури и в тези програми:
 - спецификациите не отразяват подреждането на команди, а зависимостите между данните (изпълнението на програмата е като решаване на кръстословица)
 - променливите са с еднократно присвояване
 - всички инструкции с готови операнди могат да се изпълняват едновременно (асинхронно) – в зависимост от наличните ресурси на изпълнителната платформа
 - паралелизмът е на инструкционно ниво

Потокови изчисления

9.5

- потоковите системи са с управление по вход (data driven, greedy evaluation) и управление по изход (demand driven, lazy evaluation)
- при управление по изход разборът на програмата се прави от крайния резултат в посока идентифициране на необходимите междинни резултати в програмата – докато се стигне до израз, чиито аргументи са готови; след това обработката се извършва в обратен ред на обхождането
- при управление по вход всички изчисления се извършват веднага щом необходимите им операнди са готови – потока на обработката не се анализира предварително, поради което стартирането на програмата е по-бързо, но изпълнението може да се забави ако в кода има ненужни междинни (и крайни) резултати
- потоковите езици обикновено имплицират правилото за еднократно присвояване на стойност на всяка активна променлива (което значително улеснява паралелизма, поради елиминиране на всички зависимости, които не са непреодолими)

Пример на потокова програма

9.7

- изчисление на израза $X = B2 - 4 * A * C$
 - с императивен език кодирането е примерно (9.7.1):


```
A = 1           /* step 1
B = -2
C = 1
T1 = A*C = 1
T2 = 4*T1 = 4   /* can be T1 = 4*T1 - multiple assignments
T3 = B**2 = 4
X = T3 - T2 = 0 /* step 7
```
 - с потоков език кодирането ще отразява следните стъпки (9.7.2):


```
A = 1; B = -2; C = 1 /* step 1
T1 = A*C = 1; T3 = B**2 = 4
T2 = 4*T1 = 4       /* single assignments only
X = T3 - T2 = 0     /* step 4
```

Потоково и функционално програмиране

9.4

- особеностите на езиците за функционално (апликативно) програмиране са:
 - програмата представлява дефиниция на израз или на функция върху променливи и други функционални дефиниции
 - изпълнението на функциите произвежда нови стойности без да променя тези на променливите-аргументи
- функционалното и потоковото програмиране се разглеждат като взаимни инверсии – което резултира в хибриден език за функционално паралелно програмиране SISAL
- SISAL съчетава предимствата на функционалното и потоковото програмиране като постига добра производителност на генерирания изпълним код
- VAL е потоков език, ориентиран към приложения за потокови архитектури

Потокова програма

9.6

- потоковата програма е формално описание на обработката като мрежа, която отразява зависимостите между данните, а операциите които се извършват върху тях са по-скоро като маркировка на възлите – последователността, в която ще се изпълняват инструкциите, не се задава явно и може да варира в зависимост от самите данни и системата (планирането и ресурсите)
- изчислителният граф на потоковата програма е по същество граф на зависимостта по данни, възлите на който отразяват операциите (или процесорите) а дъгите – маршрута на данните
- възможна е трансляция от командна към потокова програма и обратно
- в командната програма също може да се укаже явно паралелизъм (със съответните езици) или да се ползва паралелен компилатор; смята се, че най-добър резултат като ефективност и следователно като скорост на обработка може да се постигне с явно задаване на паралелизма на императивен език, но потоковите програмиране е сериозен конкурент

Потокови езици

9.8

- потоковата програма се описват с потоков граф, чиито възли (actors) се състоят от няколко полета – операция, едно или повече полета за входящи марки-данни (tokens) и поле за наследник на резултата;
- възлите се отнасят към някой от петте шаблона-примитиви (9.8):
 - функция (изпълнява примитивна операция върху входовете и предава резултата към изхода/ите)
 - ключ (gate – входът се предава на изхода при изпълнен предикатен израз)
 - генератор (предава константа към изхода)
 - стохастично сливане (първият готов вход се предава на изхода)
 - реплика (копие – размножава стойността на входа към изходите)
- потоковите езици са функционални (апликативни) езици; характерни представители на този клас езици са ID – на University of California - Irvin [Arvind, 1978] и VAL – Value Oriented Algorithmic Language на MIT [Ackerman, 1979]

Потокови езици - особености

9.9

- еднократно присвояване – т.е. именуване на стойности (вместо на адреси) – по принцип имената на променливи получават стойност само веднъж – включително и структури; не се допускат изрази от типа
`A := A*B` `/* illegal`
- локалност – обхвата на променливите е ограничен; няма странични ефекти – напр. изпълнението на една операция не влияе върху резултата на други операции; освен това отсъства глобално адресно пространство или памет с общ достъп
- потоките езици са априкативни – ориентирани са към генериране на стойности, които се използват за изчисление на нови стойности – до изчелпване на планираните операции
- ограничено ползване на итерации
- отсъствие на синоними – напр. не се допуска многократно позоваване на един реален параметър в списъка формални параметри на функциите от типа `MUL(A, A)` за изчисление на A^2

Интерпретация на потоките езици

9.11

- възлите се представят като структури в паметта; потоките процесори (вж л-я 1.) се състоят от команден интерпретатор CPU и пул от процесорни елементи ПЕ
- всяка инструкция представлява изпълним възел и е съчетание от операция, аргументи и адреси за резултата
- при изпълнението на инструкция ПЕ генерира пакет със стойността на резултата, адресите за предаване и евентуално марки; пакета се записва от ПЕ в памет за итерациите (memory update system) с паралелен достъп
- неизпълнените инструкции-възли, които получат стойностите си (след проверка на съответствието на марките) се предават на системата за зареждане, която планира изпълнението им от ПЕ

Съставни типове данни във VAL

9.13

- съставните типове са масиви, записи и изреждане (обединение)
- при декларацията на масивите се задава името, типа елементи и дименсията, но не и размера (той се фиксира при присвояването) – примери:
`type ARR_TYPE = array[INTEGER];` `/* type definition`
`type ARR_TYPE1 = array[array[INTEGER];`
`[1:Expression1; 2:Expression2];` `/* elements' assignment`
`/* e.g. in arr={1,2,3,4}: arr[3:6] & arr[6:7] →`
`arr={1,2,6,4,miss_elt,7}`
- аналогично се дефинират записите:
`type REC_TYPE = record[FIELDS];` `/* type definition`
`rec[A,B: INTEGER; C:REAL; D:CHARACTER; E:BOLEAN];`
`rec[A:1; B:2; C:3.14; D:'y'; E:FALSE]` `/* elements' assignment`
`F := (rec.A = 1)` `/* F = TRUE`

Паралелни изрази във VAL

9.15

- паралелното изпълнение на изрази се задава по някой/и от следните способности:
- обхват на индексите на елементите от структура за паралелна обработка
- комбиниране на резултатите от паралелно изпълнените блокове
- за целта използва конструкцията `forall` със следния синтаксис
`forall_expression ::=`
`forall name in [expression] {, name in [expression]}/* range`
`forall_body`
`endall`
`forall_body ::=`
`construct expression | eval forall_op expression /* combination`
`forall_op`
`PLUS | TIMES | MIN | MAX | OR | AND`

Дисциплина на възлите в потоките езици

9.10

- работата на възлите се определя от наличието на съответните входни стойности (tokens) – при готовност се стартира предвидената обработка на входните стойности (node firing) и резултатите се предават към следващи възли по съответните дъги, след което възелът е в престои до следващото «запалване» (9.10)
- схеми на активиране («запалване») на възлите:
- статична активация – възелът се активира когато всичките му входни дъги са готови с данни и всичките му изходни дъги са празни (за което възелът-наследник на данните изпраща потвърждение на родителя, че данните са приети)
- динамична активация – достатъчна е готовността на входовете, готовност на изходите не се изисква – поради възможността на натрупване на данни в дъгите, отделните стойности (tokens) се придружават от марки (tags) на поредността, принадлежността към определен набор данни а също и времето на генерация и възела-източник

VAL (Value-oriented Algorithmic Language)

9.12

- записите на този език съдържат неявно описание на алгоритмичния паралелизъм и при интерпретация се представят като поток граф, чрез който се планират възлите-изрази за паралелно изпълнение
- записите съдържат изрази и функции с техните входни стойности, като спецификациите трябва да гарантират принципа за отсъствие на страничен ефект от изпълнението им; двойките стойност-име получават стойност само веднъж в рамките на обхвата им (функция или блок)
- типизирането на променливите е стриктно и явно; допустимите скаларни типове са цял, реален, символен и булев:
`A: REAL := 0`
`B: INTEGER := 0`
`C: CHARACTER := '0'`
`D: BOOLEAN := TRUE` `/* operations: and, or, not, equal, not equal`
- освен основните стойности TURE и FALSE, булевите променливи могат да получават и стойности за изключения: `Undef[BOOLEAN]` и `Miss_elt[BOOLEAN]`, които съответстват на недефинирана стойност или отсъстващ аргумент
-

Изрази и функции във VAL

9.14

- дефиниция на функция:
`function Class (Param: BOOLEAN returns INTEGER); /* function def`
`initiation`
`{body – block}`
`endfun`
- в блока (тялото) на функцията са достъпни нейните формални параметри и променливите с локални дефиниции (с обхват до връщане на стойността на функцията)
- блокът `let-in` се използва за дефиниране на променливи в тялото на функцията; стойността или стойностите, които връща функцията (във VAL връщаните стойности могат да бъдат повече от една) се записват като списък с разделител запетая:
`function Calc (A, B, C: INTEGER returns INTEGER, BOOLEAN);`
`let`
`X: INTEGER := (A + B/C);`
`Y: BOOLEAN := (C !=);`
`in`
`X, Y` `/* list of`
`return expressions`
`endlet`
`endfun`

Примери за паралелни изрази във VAL

9.16

- паралелна обработка на първите пет елемента на масива `Calc`:
`forall Calc in [1, 5]`
- паралелна обработка на първите пет елемента на масива `Calc` със запис на резултата (в случая квадратите на елементите) в масив (със стойности 1, 4, 9, 16, 25):
`forall Calc in [1, 5]`
`construct Calc * Calc`
`endall`
- паралелна обработка на първите пет елемента на масива `Calc` и връщане на един резултат съгласно зададена операция (в случая $5 = 1 + 4 + 9 + 16 + 25$):
`forall Calc in [1, 5]`
`eval PLUS Calc * Calc`
`endall`

Условия и цикли във VAL

9.17

- синтаксисът на структурата за условно изпълнение if-then-else е следния:

```
Condition ::= if expression then expression
            {else if expression then expression}
            else expression
            endif
```
- структурата за цикъл-итерация е for-iter се използва за деклариране на циклични операции при зависимост по данни между последователните итерации; това е изключение от правилото за еднократно присвояване, цикълът е без формални управляващи променливи, многократната модификация на променливите може да се извърши в iter блока
- пример за изчисление на N!:

```
for I : INTEGER := 1;          /* initialization part
  P : INTEGER := N;
do                               /* loop part
  if P > 1 then
    iter
    I := I * P; P := P - 1; enditer
  else I
endif   endfor
```

Езикови принципи на функционално програмиране

9.19

- принципите, на които се базират езиките за ФП, са разработени от автора на Фортран John Backus през 70те години на ХХ век и в резултат той е предложил езика FP (Functional Programming language – 1978), в който са заложени елементите на математическите функции:
- функционални примитиви, които се елементи на езика (и съответстват на вградените или библиотечни операции в императивните езици)
- функционални форми – процедури, които представляват комбинация от примитиви
- операции на приложението – свързват функциите с техните аргументи и извличат резултата
- обекти данни – стандартизираните структури, обхват на валидност и дефиниционни области
- дефиниции на имена – метод за именуване на функциите, с който се избягва многократно повтаряне на функционални дефиниции в програмата
- както в математиката функцията и тук е изображение на наредена n-торка, чиято стойност се използва като аргумент за следваща функция в програмата

Функционални форми

9.21

- композиция на две функции: $(f \circ g):X \equiv f:(g:X)$
- конструкция на n функции: $[f1, f2, \dots, fn]:X \equiv \langle f1:X, f2:X, \dots, fn:X \rangle$ - пример: $[min, max, avg]:\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 5, 3 \rangle$
- α -обобщение (apply_to_all): $\alpha f:\langle x1, x2, \dots, xn \rangle = \langle f:x1, f:x2, \dots, f:xn \rangle$
- включване: $/f:\langle x1, x2, \dots, xn \rangle = \langle x1, /f:\langle x2, \dots, xn \rangle$ - пример: $/+:\langle 1, 2, 3, 4, 5 \rangle = 15$

SISAL

9.23

- SISAL (Stream and Iterations in Single Assignment Language) е типизиран функционален език с общо предназначение и за ефективни научни изчисления, базиран на синтаксиса на Паскал и произведен на езика VAL
- програмата на Сисал се състои от компилационни модули (разделна компилация), всеки от които е набор от функции с интерфейс за външен достъп до тях (може да има и функции само с вътрешен достъп)
- аргументите на функциите (нула или повече на брой), както и стойността на резултата (поне една) са от предварително деклариран тип (в декларативно поле – header – на компилационния модул)
- функциите са резервиран достъп до аргументите си – без странични ефекти, без псевдоними, с еднократно присвояване и именуване на стойности, а не на адреси в паметта
- тези свойства улесняват компилацията на езиковия код до езиково-независима форма на потоков граф
- средата за изпълнение на Сисал-програми инкорпорира оптимизация за паралелно изпълнение на кода, а производителността е съпоставима с тази на код на Фортран

Функционално програмиране

9.18

- функционалното програмиране е близко по съдържание и форма до потокото програмиране и също е средство за специфициране на паралелна интерпретация с представяне на изчисленията в последователна форма – функционалните езици се разглеждат като хибридация на императивните и потоковите
- поддържа се принципа за еднократно присвояване на променливите, който елиминира преодолителните зависимости и улеснява паралелната интерпретация (тъй като няма нужда от глобален анализ на зависимостите)
- програмата се състои от дефиниции на функции и изрази върху техните стойности без значение на реда на обработка на тези функции
- функциите на подреждане/планиране на операциите, комуникациите (в смисъл обмен на данни) и синхронизацията са изнесени към компилатора и интерпретиращата инфраструктура (архитектура, ОС) – възможност, дължаща се именно на улеснената идентификация на паралелните процеси (чийто паралелизъм не се задава явно от програмиста)
- паралелизма се открива динамично (вместо да се дефинира статично), същото важи и за обмена и синхронизацията
- в резултат функционалната програма е еднаква за различен тип и клас архитектури и самите езици не предвиждат специализирани средства за спецификация на паралелизъм, синхронизация и т.н.
-

Функционални примитиви

9.20

- примитиви за избор са FIRST, LAST и TAIL:
 $x1 \leftarrow FIRST(x1, x2, \dots, xn)$
 $xn \leftarrow LAST(x1, x2, \dots, xn)$
 $\langle x2, \dots, xn \rangle \leftarrow FIRST(x1, x2, \dots, xn)$
- примитивите за структуриране ROTR, ROTL, LENGTH и CONS се използват за структурни операции върху елементите:
 $\langle xn, x1, \dots, xn-1 \rangle \leftarrow ROTR(x1, x2, \dots, xn)$
 $\langle x2, \dots, xn, x1 \rangle \leftarrow ROTL(x1, x2, \dots, xn)$
 $n \leftarrow LENGTH(x1, x2, \dots, xn)$
 $\langle x, x1, \dots, xn \rangle \leftarrow CONS(x, \langle x1, x2, \dots, xn \rangle)$
- аритметични бинарни операции: +, -, *, div и | (за остатък):
 $residue_x1_by_x2 \leftarrow |:\langle x1, x2 \rangle$
- предикатни операции със стойност Т или F
- логически операции върху булеви аргументи
- операция за идентичност: $x \leftarrow ID:x$

Езици за функционално програмиране

9.22

- FP не е развил достатъчно изрази и операционни средства, но разработва принципите на други езици за ФП като Лисп, Странд и Сисал
- STRAND (Stream And) е език за ФП с поддържане на потокови данни (streams – не в специфичния смисъл на мултимедийните комуникации, а като непрекъснат поток на обмен между конкурентни задания) и на AND-паралелизъм (информационно-свързаните задания се изпълняват паралелно) – възприема редица принципи на потокото програмиране
- Странд-приложенията са преносими (и ефективни) на еднопроцесорни и паралелни компютри (от различен тип)
- програмите на Сисал са структури от функции и математически изрази, чието изпълнение ангажира променлив брой от процесори с неявно задаване на паралелизма; стойностите имат имена и обработката не съхранява (и не се нуждае) от статичен контекст
- програмите на Сисал се транслират до потокови графи, които имат машиннонезависима интерпретация и съдържат зависимостта по данни между операциите

Типове данни в SISAL

9.24

- SISAL поддържа скаларните типове цял, реален, символен, булев и двоен, както и структурите масив, запис, обединение (union) и поток (stream)
- масивите еднодимензионни (или масиви от масиви) с определен тип за всички елементи и могат да имат различен долен индекс и размер:

```
type OneDim = array[integer]
type TwoDim = array[OneDim]
X := array OneDim[ ]                               /* empty array of integers
Y := array OneDim[1: 1, 2, 3]                       /* low index 1
Z := array TwoDim[1:array[1: 1, 2, 3], array[1: 7, 8, 9]
```
- поток е последователност от наредени елементи-стойности с еднакъв тип и достъпни по реда на предаване (не с произволен достъп), с един източник и един или повече получатели
- елементите на потока са достъпни за получателите веднага щом бъдат създадени от източника и се използват за конвейерен паралелизъм или за В/И

Паралелни изрази в SISAL

9.25

- паралелизмът не се задава явно и няма специализирана езикова поддръжка (което прави кода универсално преносим)
- възможността за паралелно изпълнение на циклите се поддържа от следните блокове
 - for initial: допуска паралелно изпълнение на итерациите с обръщение към стойности, дефинирани в други итерации; състои се от четири компонента:
 - инициализация – зарежда управляващите променливи на цикъла и стойностите на останалите променливи
 - тяло на цикъла – ключът на тялото на цикъла може да е префиксен (while) или постфиксен (until) с обичайната семантика и възможност тялото да не се изпълни нито веднъж в първия случай
 - проверка за край – изчислява новите стойности на управляващите променливи (отклонение от принципа за еднократно присвояване – за синтактична съвместимост старата и новата стойност на управляващите променливи се разграничават с префикса old
 - клауза за резултатите – резултатът е крайната стойност на името на съответния цикъл или редуция от всички крайни стойности на итерациите, която се задава с някоя от следните седем клаузи: array of, stream of, catenate, sum, product, least, gratest
 - for: за независими итерации без обмен на данни; състои се от три компонента:
 - генератор на обхвата – определя размера и композицията на агрегирани (съставни) структури като резултат от dot или cross операция
 - тяло на цикъла – набор от изрази за всеки елемент
 - клауза за резултатите – като при for initial

SISAL компилатор

9.27

- компилаторът на Сисал Osc е със сложна структура тъй като трансляцията от абстрактните спецификации на Сисал до обектен код за съответната паралелна архитектура се извършва в седем фази
- трансляцията от Сисал към междинната форма IF1 се извършва от парсер като резултат е ациклически поток граф с функционална семантика: възлите са или елементарни операции (вкл. манипулции на масиви и потоци) или съставни възли, съдържащи подграфи
- свързането с библиотечен код се осъществява от IF1LD, началната машиннонезависима оптимизация – от IF1OPT
- в следващите фази са за статична алокация на адресите на масиви и други структури, след което се извършва проверка на паралелизма от IF2PART, който определя граниларността и извършва разделянето на паралелни подзадания
- последната фаза е CGEN, която извършва трансляцията от IF2 към Си код, подлежащ на локална компилация, и също генерира синхронизационните примитиви за съответната платформа; Си осигурява преносимост и възможност за допълнително настройка на генерирания код

Особености на програмирането за SIMD

9.29

- в SIMD една и съща инструкция се изпълнява върху различни данни от отделните процесорни елементи; паралелизма е на инструкционно ниво (контраст с SPMD); синхронизацията е апаратно-вградена
- N.B.: паралелната обработка на данни (ПОД) като правило поражда много по-високо ниво на паралелизма отколкото паралелизма по управление (който обикновено е някаква форма на конвейеризация) – дори и когато последния се прилага при същата фина грануларност – на ниво инструкция; по-високият паралелизъм обаче не означава автоматично и по-добра ефективност
- поради синхронното изпълнение на паралелните инструкции не се налага приложението на специални езикови средства за управление на синхронизацията и паралелизма като цяло и програмата може да се специфицира и с конвенционален език; пример – C-код за векторни изчисления (9.29):

```
for (i = 0; i <= N; i++) {
    A[i] = A[i] + K;
    B[i] = B[i] * A[i];
}
```

ПОД за MIMD

9.31

- MIMD архитектурите са с по-големи операционни възможности от SIMD, поради което могат да интерпретират ЕПОД-програми, но когато интерпретацията е директна, това налага съответно и най-фина грануларност – на инструкционно ниво, което обикновено не е най-ефективния режим на работа на MIMD машините
- по-рационално следователно е да се изостави изискването за синхронно изпълнение на отделните инструкции, като точките на синхронизация се запазват само при операциите за междупроцесорен обмен на данни – резултата очевидно е SPMD-модел на обработка
- макар че MIMD са пригодени за изпълнение и на паралелизъм по управление, обикновено се предпочита приложението им в SPMD-режим винаги когато това е възможно (в зависимост от паралелния алгоритъм) – по-подробно за ПОД в MIMD архитектури

Примери за паралелни изрази в SISAL

9.26

```
for initial
  I := 1;
  X := Y[1];
  while I < N repeat
    I := old I + 1;
    X := old X + Y[I];
  returns array of X
end for

for I in 1, N
  X := A[I] * B[I]
  returns value of sum X
end for

for I in 1, N cross J in 1, M /* array NxM as
  aggregate of 2 sequences
  returns array of (I * J)
end for
```

SISAL ядро

9.28

- Сисал изисква системна инфраструктура – ядро, което да изпълнява генерираните многозадачни приложения, изпълнявайки функциите по динамично планиране на паметта и интерфейс към ОС за вход-изход и командна интерпретация
- при наличие на p процесора (където броя процесори се задава с атрибут на командата за стартиране на приложението) всъщност може да не съответства на актуалния брой процесори в архитектурата) ядрото разделя циклите на p части и ги обособява като отделни нишки (или леки процеси) в специална опашка, от където се извличат за изпълнение
- еталонни програми като комбинираните тестове за научни изчисления Livermore Loops (24 изчислителни алгоритъма вкл. елиминацията на Гаус-Журдан) показват съпоставимост на производителността на Фортран и Сисал върху еднопроцесорни архитектури и ускорения между 7.3 и 9.0 върху 10-процесорна SISD архитектура (Cray Y-MP – 1992)

Езикови разширения за ПОД

9.30

- по-големи възможности за изразяване на паралелизма при SIMD обработка все пак се постигат със специализирани диалекти на конвенционалните езици – напр. C* и FORTRAN90 – известни като data-parallel languages (тук: езици за паралелна обработка на данни, ЕПОД)
- все пак спонтанността, с която се изразява паралелизма при SIMD, се нуждае от сериозна системна поддръжка – не за синхронизацията и управлението на потока инструкции – а за планиране и разпределяне (mapping) на паралелно изпълняваните инструкции върху отделните ПЕ; тази поддръжка е статична по своя характер, поради което е по-ефективно да бъде изпълнена от ЕПОД-компилаторите – каквато е и обичайната практика при системното осигуряване на SIMD
- по-конкретно специфичните функции на ЕПОД-компилаторите са:
 - разпределяне на ПЕ за паралелните инструкции,
 - планиране на паметта за паралелен достъп
 - планиране на междупроцесните комуникации и
 - добавяне на инструкции за главния процесор, осигуряващи паралелното зареждане според извършеното разпределяне

ПОД със C*

9.32

- C* е език за ПОД със разширен синтаксис на стандартния C и елементи на ООП като в C++, който представя изпълнителната архитектура като интерфейс или главен (front-end) унипроцесор, разширен с ко-процесори (ПЕ) за SIMD обработка (back-end) – 9.32
- типовите данни, операторите, конструкциите, указателите и функциите са наследени от C (+ съответните езикови разширения) и операциите върху скалари се изпълняват от главния процесор по начин, по който би се изпълнил стандартен C-код
- програмите следват класическото императивно (control-flow) управление и изпълняват векторните инструкции върху векторните ПЕ, чието локално адресно пространство е достъпно за главния процесор
- броя и топологията на ПЕ са динамично настройваеми (т.е. по време на изпълнение на програмата)
- програмата се състои от последователни блокове за паралелно (domain – върху ПЕ) и последователно изпълнение (само върху главния процесор)
- данните са скаларни (декларират се като topo и се зареждат в паметта на главния процесор) или векторни – roly, които се разпределят в локалните паметти на ПЕ
- трансляцията към паралелен код се извършва от компилатора на C*, който преобразува стандартна скаларна операция за паралелно изпълнение върху данните в ПЕ
- C* е разширен с израз за селекция, който активира съответния брой ПЕ за изпълнение на векторните операции
-

Шаблони за паралелни данни в C* 9.33

- ▶ паралелните променливи се разполагат в ПЕ за векторна обработка (в зависимост от съотношението между размерите на вектора и на системата)
- ▶ атрибут на паралелните променливи е `shape` – шаблон, който задава мощността и структурата на паралелната променлива – като стандартен набор от еднотипни скаларни елементи – с което се заявява паралелна обработка на съответната променлива:

```
shape [10][10] array; /* 10x10 template
shape [5][5][10] cube_array; /* total 250 elements
```
- ▶ шаблонът се характеризира с брой дименсии или оси (ранг), но няма специфичен тип
- ▶ паралелните променливи се задават с деклариран шаблон и тип:

```
shape [10][10] array;
shape [5][5][10] cube_array;
int: array array1; /* parallel variable array1 of 100 integers
int: cube_array grade[100]; /* grade: 250 elements of 100
integers each
```

Паралелни операции в C* 9.35

- ▶ когато поне един от операндите е деклариран като паралелна променлива, операцията се изпълнява паралелно, за което е необходимо:
 - ▶ операндите да са със съвместими шаблони за съответната операция – напр. масиви с еднакъв размер и дименсия
 - ▶ операцията да е зададена с израз `with`, който зарежда съответния контекст в ПЕ
- ▶ пример:

```
shape [10][10] array;
integer: array x, y; /* two similar arrays of integers
with (array) {
x = x + y; } /* element-wise addition
```

Паралелни операции върху подмножества елементи 9.37

- ▶ изразът `where` с `0` п `c` и `1` дефинира подмножества от елементите на паралелни структури – “активни позиции” – върху които се извършва обща паралелна операция:

```
shape [10][10] array;
integer: array x, y;
with (array) {
where (y <> 0) {
x = x/y; /* active positions of
positives only
else
x = Max_int; } } /* non-positives only; blue
code optional
```

Елементи на ЕПОД FORTRAN90 9.39

- ▶ FORTRAN90 е ЕПОД, който разширява стандартния фортран с указатели, потребителски типове рекурсия, динамична алокация на памет, функции за обработка на масиви и др. – генерации фортран 9.39.1
- ▶ програмният модел, върху който се изпълнява този код, включва централен процесор, логическо устройство за скаларна аритметика и такова за векторна обработка и обща памет – 9.39.2
- ▶ последователните инструкции се изпълняват от главния процесор, който управлява и работата на двата аритметични копроцесора
- ▶ опреациите с векторните променливи се специфицират като скаларни, но обработката им се извършва паралелно и синхронно – т.е. на езиково ниво паралелизма е имплицитен

Шаблонни обръщения в C* 9.34

- ▶ обръщението към елементите на шаблона е с ляво единично индексирание, което съответства на алокацията им в ПЕ: `[0]array1` – елемента в първия процесор
- ▶ шаблоните паралелни променливи могат да бъдат съставени и от C-структури:

```
shape [10][10] array;
struct list {
int id;
float income;
char* name; }
struct list: array listA; /* 100 elements of type
list in shape array
```

като компонентите на структурата са достъпни със стандартния запис `[15]listA.id`

Редукции в C* 9.36

- ▶ C* дефинира набор от вградени оператори (редукции), с които основни операции върху шаблонни паралелни операнди, чиито резултат е скаларен, могат да се представят (езиково) като последователни операции; получените от редукцията скалари могат да се използват и неявно в стандартни C функции:

Оператор	Резултат
<code>+=</code>	скаларна сума на елементите на паралелна шаблонна променлива
<code>-=</code>	негативна сума на елементите
<code>&=</code>	побитова конюнкция на елементите
<code>^=</code>	побитово изключващо ИЛИ на елементите
<code> =</code>	побитова дизюнкция на елементите
<code>>?=</code>	максимална стойност на елементите
<code><?=</code>	минимална стойност на елементите

- ▶ пример:

```
integer total;
with (array) {
total = (+ x); }
printf("The maximal element is %d: ", >?= x); /*
implicit scalar
```

Комуникации в C* 9.38

- ▶ обмяна на данни между ПЕ в C* може да бъде решетъчен (“grid”) когато се извършва обмен между елементи от паралелни променливи с общ шаблон, или обща – когато шаблоните са различни
- ▶ решетъчният обмен се извършва с функцията `pcoord`, която извършва пренос на елементите на фиксирано отместване по съответната ос:

```
source2 = [pcoord(0)+1][pcoord(0)+1]source1
```

source2:	source1:
1 2 3	
4 5 6	1 2
7 8 9	4 5
- ▶ общият обмен се извършва с шаблон на преноса, който съдържа индексите на разполагане на елементите и се записва вляво от паралелната променлива на резултата – операция `send` – или вляво от паралелната променлива-източник – операция `get` (9.38):

```
[index]source2 = source1;
/* source2[index[0]] ← source1[0] ...
/* source2[index[i]] ← source1[i]
source2 = [index]source1;
/* source2[0] ← source1[index[0]] ...
/* source2[i] ← source1[index[i]]
```

Декларации във FORTRAN90 9.40

- ▶ синтаксисът при декларацията на променливи е

```
type [(kind)] [, attribute]... :: variable_list
```

където
- ▶ `type` е валиден фортрански тип като `REAL`, `INTEGER`, `CHARACTER`, `LOGICAL`
- ▶ `(kind)` е опция, която допълнително дефинира стандартния тип – напр. `CHARACTER (LEN = 10) ::...` задава максималната дължина на символен низ
- ▶ `[, attribute]` е списък-опция с водеща запетая и разделител запетая, който съдържа стандартни атрибути на променливите
- ▶ променливите са със стандартен формат на имената и разделител запетая

Изрази върху масиви във FORTRAN90^{9.41}

→ декларацията на променливи-масиви се прави с атрибута `DIMENSION`, чиито аргументи указват броя дименсии и техните граници:

```
INTEGER, DIMENSION (1 : 10) :: int_vector
```

→ операциите с масиви могат да се запишат като операции със скалари, но контекста задава паралелна интерпретация:

```
/* FORTRAN77          /* FORTRAN90
INTEGER A(10), B(10), C      INTEGER C
DO I = 1, 10, 1              INTEGER, DIMENSION ( ) :: A,
  A(I) = B(I) + C            B
END DO                      A = B + C
```

→ могат да се задават области и селекции от масиви като се използва записа:

```
V(lower_bound : upper_bound : stride) /* stride is optional selection
/* and can be negative as well
```

→ например:

```
• INTEGER, DIMENSION (1 : 10) :: A, B, C
• A(1 : 5) /* first five elements of A
• A(1 : 10 : 2) /* all elements with odd indices
• A(1 : 5) = B(1 : 5) + C(2 : 6) /* non-corresponding
  subscripts
```

Вградени ("intrinsic") функции върху масиви във FORTRAN90^{9.43}

→ библиотеката с вградени функции върху масиви не се нуждае от явно деклариране в програмата, машинния код за тези функции се добавя автоматично на етапа свързване

→ няма синтактично разграничаване между наследените функции за скалари и домавените функции върху масиви – отново контекста задава типа операция имплицитно

→ някои вградени функции:

функция	стойност
MAXVAL (A)	максимален елемент – стойност
MINVAL (A)	минимален елемент – стойност
MAXLOC (A)	максимален елемент - позиция
SUM (A)	сума на елементите
PRODUCT (A)	произведение на елементите
MATMUL (A, B)	матрично произведение
DOT_PRODUCT (A, B)	произведение на матрица и скалар
TRANSPOSE (A)	транспониране
CSHIFT (A, SHIFT, DIM)	ротация на елементите (SHIFT >0 → надясно)

10.2

Съдържание

- Процеси и нишки
- Мултипроцесинг в UNIX
- Миграция на код
- Идентификация на обекти
- Garbage collection

10.4

Паралелни процеси

→ паралелизма (грануларността) е на ниво програма и процедура

→ това ниво съответства на мултипроцесинга, тъй като всяка програма е отделен процес

→ при SPMD модел (напр. в UNIX) с примитива `Fork` се създава реплика на изпълняващия процес:

```
Proc-id = Fork()
(създава се нова реплика на процеса и ѝ се присвоява идентификатор)
```

→ двата процеса (родител и наследник) се различават само по стойността на `Proc-id`, в наследника тя е 0:

```
Proc-id = Fork()
if Proc-id = 0
  then do {child processing}
  else do {parent processing}
```

→ други примитиви от тази група са `exit` за прекратяване на процеса наследник и `wait` за синхронизация (процесът-родител блокира до завършване изпълнението на наследника)

→ при процедурен паралелизъм на системно ниво процедурата се асоциира с отделен процес

```
Proc-id = new process(A_procedure)
kill process (A_procedure)
```

Многодименсионни масиви във FORTRAN90

9.42

→ при многодименсионните масиви селектиращите операции върху отделните оси се разделят със запетая:

```
INTEGER, DIMENSION (1 : 3, 1 : 6) :: A /* 3 rows by 6 columns
A(2, : ) /* all elements of row 2
A(2, 3 : 5) /* elements 3, 4 and 5 of row 2
A(2, 1 : 6 : 2) /* elements 1, 3 and 5 of row 2
```

→ конструкцията `where-elsewhere-end where` (`elsewhere` – опция) задава условна селекция:

```
INTEGER, DIMENSION (1 : 3, 1 : 6) :: A
where (A > 0) A = sqrt(A) /* root takes positives
only
```

10.1

10. Управление на процесите в разпределени системи

10.3

Процеси

→ в ОС процесите са системни и потребителски програми, допуснати до изпълнение, за които динамично се планират необходимите операционни (процесорно време, памет, В/И и др.) и комуникационни ресурси

→ планирайки, ОС създава за всеки процес виртуален процесор и позиция в таблицата на процесите с регистърен буфер, карта на процесната памет и на отворените файлове, приоритети, процесно счетоводство и др. – също и за междупроцесна защита

→ създаването/превключването на процеси (процесен контекст) е сериозен системен свръхтовар – напр.:

- алокация на сегмент за данни (евентуално нулиран)
- зареждане на кодовия сегмент, алокация/зареждане на стека, на регистрите (процесорни `r`-ри, програмен брояч, стеков указател, MMU и TLB регистри)
- управление на `swap` операции между основната и външната памет (при мултипрограмиране с повече процеси)

10.5

Паралелизъм на ниво израз

→ това ниво е свързано с езикови спецификации (примитиви за паралелно изпълнение на инструкции)

→ напр. примитивът `Parbegin/Parend` задава блок от изрази за паралелно изпълнение, по време на което главният процес блокира

→ пример – изразът $(a+b)*(c+d)-e/f$ може да бъде изпълнен конкурентно със следната спецификация (псевдокод):

```
Parbegin
  Parbegin
    t1 = a + b
    t2 = c + d
  Parend
  t4 = t1 * t2
  t3 = e/f
Parend
t5 = t4 - t3
```

Паралелен израз в UNIX

- ▶ паралелизъм на израз с примитивите `fork-join-quit`:
- ▶ `fork` label предизвиква създаване на нов процес-наследник, чието изпълнение стартира от даден етикет (така наследника и родителя може да не са реплики):
- ▶ `quit` е примитив за прекратяване на текущия процес
- ▶ `join t, lab` е примитив със следната интерпретация:


```
t = t - 1
if t = 0 then go to lab
```
- ▶ пример за изчисление на горния израз


```
n = 2
m = 2
Fork P2
Fork P3
P1: t1 = a + b; Join m, P4; Quit;
P2: t2 = c + d; Join m, P4; Quit;
P4: t4 = t1 * t2; Join n, P5; Quit;
P3: t3 = e/f; Join n, P5; Quit;
P5: t5 = t4 - t3
```

10.8

Многопроцесно приложение в UNIX

- ▶ за вътрешна идентификация на процесите често се прилага и индексирание: пример – функция `Mkps()` за създаване на n процеса-наследници със стойност на `ppid` 0 в процеса родител и от 1 до n в наследниците:


```
mkps(n)
{
  int n;
  {
    int i;
    for (i = 0; i < n; i++)
    {
      switch (fork())
      {
        case 0: /* process creation*/
          return(i+1);
        case -1: /* cannot create process*/
          {
            printf("cannot create process %d\n", i);
            return -1; }
          default: /* goto next creation */
            }
      }
    }
  return 0;
}
```

10.10

Обмен между процесите

- ▶ UNIX няма други средства за деклариране на общи ресурси между потребителските процеси освен общи променливи, чиито тип зависи от използвания език
- ▶ променливата или структурата, която е с общ достъп, се декларира съгласно езиковия стандарт:


```
struct SharedData
{
  int x, y, z;
  float a, b;
  char* name
} mySharedRecord, *toMySharedStruct;
```
- ▶ всяка [вече] декларирана променлива може да бъде обявена за общ достъп (и алоцирана в общ сегмент от паметта) със системната заявка `Share()`: `toMySharedStruct = Share(0, sizeof(mySharedRecord));`
- ▶ резултатът е, че освен деклариращия процес, всички негови наследници (създадени след нейната декларация като обща) имат достъп до съответната променлива

10.12

Синхронизация с взаимно изключване между процесите в UNIX

- ▶ променливи от тип `lock` осигуряват монополен достъп на извършваните върху тях операции за даден процес
- ▶ специфичните операции за този тип са `lockname_create` и `lockname_init`, където `lockname` е множество от следните типове: `LOCK`, `BARRIER`, `SEMAPHORE` и `EVENT`
- ▶ `LOCK` е тип данни, с който е асоцииран атрибут със стойности `PAR_LOCKED` и `PAR_UNLOCKED` и се реализира класическия подход за взаимно изключване: с този тип са асоциирани и операциите `lockname_lock` и `lockname_unlock`
- ▶ `BARRIER` е тип данни, съставен от двойката (`count`, `flag`), където `count` задава броя процеси, чието изпълнение трябва да достигне до съответния обект-барьера, преди да продължат; `flag` задава режима на изчакване
- ▶ `flag = SPIN_BLOCK`: блокировка с циклично изчакване
- ▶ `flag = PROCESS_BLOCK`: блокировка при достъп до данни
- ▶ `EVENT` е тип данни, съставен от двойката (`event`, `flag`), където `event` задава събитие, което трябва да се изпълни преди процеса да продължи (възможно е повече от един процес да чака това събитие); `flag` задава режима на изчакване като при `BARRIER`
- ▶ `SEMAPHORE` е тип данни, асоцииран с двойката атрибути (`count`, `flag`), където `count` задава броя процеси, които имат право на достъп до променливата преди заключването ѝ; `flag` задава режима на изчакване като при `BARRIER`; с този тип е асоциирана операцията `semaphore_set` за `count`

Паралелно програмиране в UNIX

- ▶ най-разпространената ОС за паралелни системи
- ▶ процесите се управляват чрез системни заявки (`calls`):
- ▶ създаване: използва се заявката `fork()` за репликиране на текущия процес-родител
- ▶ планиране и контрол – напр. с използване на системния таймер – функциите `timer-init()` и `timer-get()` (в микросекунди) или с използване на семафори
- ▶ междупроцесен обмен – чрез алоциране на общи променливи със заявката `Share()`
- ▶ паралелните приложения се разработват най-често на C с използване на библиотеката `parallel.h` и се компилират с опция `-lpp` за зареждане на паралелната библиотека:


```
cc program -lpp
```

10.9

Паралелно програмиране в UNIX

- ▶ шаблон на паралелната програма:


```
main (argc, argv)
int argc;
char * argv[];
{
  int ppid, procs;
  scanf(argv[1], "%d", &procs);
  ppid = Mkps(procs); /* creation of argv[1] number of processes*/
  switch (ppid)
  {
    case 0: { /* parent process code */
    case 1: { /* child1 process code */
    case 2: { /* child2 process code */
    ...
    case n: { /* childn process code */
    default:
      {
        printf("Program error"); break;
      }
    }
  } /* termination of the children: */
  if (ppid != 0)
  {
    printf("child # %d terminates\n", ppid);
    exit(ppid);
  }
}
```

10.11

Времево планиране на процесите

- ▶ времето планиране е частен случай на планирането по събитие, при който събитието е изтчане на таймер;
 - ▶ ползва се системния часовник с импулси на всяка микросекунда;
 - ▶ заявка към системата `timer-init()` стартира (нулира) локален брояч за процеса, а заявката `timer-get()` връща текущата му стойност в микросекунди
 - ▶ пример: паралелни процеси с локални променливи за времето
- ```
main (argc, argv)
int argc;
char * argv[];
{
 double ProcsTime;
 long timer;
 int ppid, procs;
 scanf(argv[1], "%d", &procs);
 ppid = Mkps(procs); /* creation of argv[1] number of processes*/
 switch (ppid)
 {
 case 0: /* parent process code */
 {
 timer-init() /* set the clock */
 timer = timer-get(); /* get current time */
 ProcsTime = (timer-get() - timer)/1000000.0;
 break; /* get execution time in sec */
 }

 }
```

10.13

## Особености на процесите в разпределените системи

- ▶ ефективното планиране на разпределените приложения (предимно по модела клиент-сървер) с прилагане на многонишков подход (`multithreading`) за припокриване (`overlapping`) примерно на комуникационните фази с фазите на локална обработка на отделните процеси;
- ▶ разлики в планирането при клиентски и сърверни машини както и между сърверите с различно предназначение (напр. обработвачи, файлови, комуникационни, за разпределени обекти и др.)
- ▶ възможности за мигриране на процеси особено в хетерогенна среда и необходимата динамична реконфигурация на клиенти и сървери (процеси)
- ▶ прилагане на обработка с процеси-агенти – равнопоставени процеси за обслужване (вместо асиметричния модел клиент-сървер)

## Нишки

- подпроцесите – традиционно «нишки» (threads) – са средство за постигане на по-фина грануларност респ. по-оптимално планиране
- при нишките е недопустим свърхтовар като при процесите → по-слаба конкурентност и защита: нишковия контекст се състои например от CPU-контекста и текущ статус (напр. блокировка поради синхронна комуникация); така че защитата на нишковите данни в рамките на процеса зависи от кодирането на многонишковото приложение (→ по-сложно програмиране)
- многонишково програмиране се прилага и при унипроцесорни приложения –
  - пример: електронна таблица с отделни нишки за потребителски интерфейс и за обработка на формулите
- многонишковата програма за унипроцесор е преносима и за паралелна обарботка
- многонишковите програми са по-удобни за настройка – пример: текстов редактор с отделни нишки за UI, граматическа проверка, форматиране, генерация на съдържание и т.н.

10.16

## Леки процеси

- LWP (lightweight process) – хибриден подход – леките процеси се изпълняват като обикновени процеси; един процес може да включва няколко LWP
- същевременно се ползват и пакетите за многонишкови програми при които нишковите операции са в потребителски режим
- многонишковите приложения създават необходимите нишки (потр. режим) и предават (имплицитно) изпълнението им на LWP – фиг. 10.16.
- LWP се създават с обръщение към системата и се асоциират с някоя от активните нишки (съгласно диспечерска процедура)
- изпълнението на "двойката" системен LWP и потребителска нишка протича предимно в потребителски режим – LWP се превключва в контекста на нишката и напр. синхронизацията с mutex се изпълнява в потр. режим
- при блокиране на нишка (поради блокиращо обръщение към системата) управлението на двоенния подпроцес се връща в режим на ядрото/LWP, а междуременно системата предава управлението на друг LWP (който ако не е блокиран, минава в режим на асоциираната с него нишка – т.е. потребителски)
- системният свърхтовар е редуциран (в потребителски многонишков режим) и изпълнението на целия процес е свободно от блокировка
- LWP са прозрачни за кода; преносимост за унипроцесорно и паралелно програмиране (във втория случай леките процеси на едно приложение се изпълняват на различни процесори)

10.18

## Многонишкови сървери

- многонишковите сървери обикновено се конструират с нишка-диспечер, която получава всяка нова заявка за обслужване и я асоциира с някоя от изчакващите изпълнителни нишки – фиг. 10.18.
- пример: при файлов и документен сървер еднонишков обслужващ процес ще изпълнява заявките последователно – вкл. и закъснението за достъп до вторичната памет
- многонишковите "диспечер-изпълнител" процеси се базират на блокиращи обръщения към системата в изпълнителните нишки

10.20

## Модели за миграция на код

- ниска (weak) мобилност – само на сегментите код и данни; изпълненито стартира отначало – пример: Java аpletите (изисквания за преместваемост на кода)
- висока (strong) мобилност – + сегмента на статуса;
- по инициатива на изпращащия процес – примери: изпращане на програма за изпълнение от изчислителен сървер (изпр. п-с е клиент; за защита е необходима идентификация на клиента) или изпращане на процес за балансиране на товара при групово обслужване (изпр. п-с е сървер)
- по инициатива на приемащия процес – Java аплети (прием. п-с е клиент) или отново за балансиране но при инициатива на приемащ сървер

## Видове нишки

- в разл. ОС се прилагат нишки в потребителски режим или в режим на ядрото
- при **нишки в потребителски режим** се ползват програмни пакети за многонишкови програми с операции за деклариране на нишките (`create`, `destroy`), за синхронизация достъпа до общи променливи – `mutex` (ключалка като семафорите с решаване на блокировката чрез приоритети или FIFO)
- по-нисък системен свърхтовар – без операции върху паметта: при създаване/закриване само заделяне и освобождаване на стека и при превключване - само замяна на стойностите в ЦПУ регистрите
- недостатък: блокирането на една нишка (напр. по ВИ) блокира целия процес – т.е. елиминира основно преимущество на многонишковия процес
- **нишките в режим на ядрото** са компоненти на системната библиотека и се изпълняват като процеси на ядрото – създаването и превключването им са с обръщения към системата – преодолява се тоталното блокиране, но свърхтовара е съпоставим с процесния

10.17

## Многонишкови клиентски процеси

- обикновено постигат маскиране на комуникационните и синхронизационни закъснения на някои нишки чрез изпълнение на други
- пример: Уеб браузерите (клиент в интерактивен режим) изобразяват веднага заредените елементи и постепенно попълват страницата – след зареждане на [част от] основната страница (най-често текст) се активира нишка за неговото изобразяване, плъзване (scroll), избор и др. функции и друга нишка/и за блокиращото зареждане на по-бавните компоненти (за блокираща заявка към ОС за връзка със съответния сървер/и);
- при повече от една комуникационна нишка се постига паралелизъм и на комуникациите/зареждането на останалите компоненти (но само ако сървера разполага със съответна производителност – напр. репликрани сървери (т.е. един адрес но реплики на страниците на няколко машини, които се асоциират прозрачно със заявките на отделните нишки напр по Round Robin)

10.19

## Миграция на код

- среща се под формата на:
  - миграция на процеси – напр. за балансиране на локалния изчислителен товар между възлите (измерван напр. с дължина на локалната опашка от заявки, натоварване на процесора/обсл. устройство и др.)
  - мигриране на програми за отдалечено изпълнение –
    - при сървера – напр. зареждане в сървера на програма за локална обработка на данни и връщане само на резултата (вместо зареждане на данните при клиента)
    - при клиента – напр. зареждане в клиента на програма за попълване параметрите на заявка и връщането ѝ към сървера (вместо интерактивен обмен със сърверен процес за попълване на заявката)
- миграцията на процес изисква преместване на сегмента код, сегмента данни и сегмента изпълнение (т.е. статус)
  - при сегмента данни: процес свързване (binding) т.е. настройка на адресните аргумент (данни); варианти:
    - свързване по идентификатор – напр. при мигриране на данни, които са адреси на файлове с URL идентификация (понеже идентификатора е унисресален)
    - свързване по стойност – напр. адресиране на стандартна библиотека в C и Java (действителния им идентификатор е локален)
    - свързване по тип – напр. адресиране на локални устройства (принтери, монитори)

10.21

## Миграция на код в хетерогенна среда

- при ниска мобилност (само на код и данни) е необходима прекомпиляция на програмата за различни машини/ОС – напр. изпращащия процес има различен изпълним код за всяка от възможните приемащи платформи
- при висока мобилност (код, данни и статус) – с поддържане на машиннонезависим миграционен стек в определени точки на програмата. (в които и само в които може да се извърши миграцията)
  - в процедурните езици (C) това е след изпълнението на текуща функция/метод и преди стартирането на следваща – за да не се налага примарно пренос на стойностите на процесорните регистри, които са машиннозависими
- с интерпретирани езици – при скриптовите езици виртуалната машина директно интерпретира програмния код (Tcl) или универсален междинен код, генериран от компилатор (Java)

## Имена, адреси и идентификатори

- имената са символни низове за идентификация на компоненти – ресурси (възли, устройства вкл. вторични паметни, файлове) и обслужвани компоненти (процеси, потребители, съобщения, документи, нюзгрупи, мрежови съединения и др.)
- именуваните компоненти подлежат на управление или промяна посредством съответни точки за достъп – адреси
- в РС са широко застъпени **динамичните адреси** → имената са по-удобни за идентификация на повечето компоненти отколкото динамичните адреси
- същото важи и за **множествените адреси** – един компонент с няколко адреса (точки за достъп) се идентифицира с име, но не и с един от адресите си; пример – разпределена Web услуга, изпълнявана от няколко сървера с различни адреси
- при имената и адресите се допуска многозначност и промяна
- за прозрачна идентификация се използват **адреснезависими имена**
- идентификаторите** са имена, които имат еднозначно-обратимо и устойчиво съответствие с компонентите:
  - всеки идентификатор съответства най-много на един компонент
  - всеки компонент има не повече от един идентификатор
  - идентификаторите не се подмянат или пренасят на други компоненти
- идентификаторите осигуряват лесно сравняване** на идентичността на компонентите (за разлика от имената и адресите поради тяхната многозначност и преходност)
- имената (когато са потребителски-ориентирани) са по-удобни за потребителите** (отколкото машинно-ориентирани идентификатори и адреси)

10.24

## Разслоено пространство на имената

- при големите/глобалните РС пространството имена се организира йерархично чрез разслояване, поддръжки общ корен
- обикновено се приема трислоен модел:
  - глобално ниво (global layer) – корена на графа и свързаните с него възли-директории; на това ниво промените на имена са много редки (най-висока стабилност), отделните възли съдържат списък с имена от следващото ниво, групирани по организационен принцип (напр. в DNS областите com, edu, gov, mil, org, net, и на страните)
  - административно ниво (administration layer) – възлите-директории съдържат списъци с компоненти, принадлежащи на обща административна област (напр. списък с отделите на една организация или списък със хостовете в даден интранет или списък на всички потребители от тази област) – относителна стабилност (в DNS sun.com, uni-sofia.bg, fmi.uni-sofia.bg, acm.org)
  - локално ниво (managerial layer) – възлите-директории представят локални компоненти – напр. файловата система на отделени хостове в дадена локална мрежа и отделни локални директории и файлове за общ достъп – ниска стабилност; поддръжката на такива възли-директории се извършва и от потребителите (в DNS courses.fmi.uni-sofia.bg)
- освен йерархично, пространството имена се разделя и административно на неприпокриващи се части – зони – всяка от които се обслужва от съответен сървер на имената

10.26

## Ресурсни записи

| [RFC1035] |                                            |
|-----------|--------------------------------------------|
| A         | 1 a host address                           |
| NS        | 2 an authoritative name server             |
| MD        | 3 a mail destination (Obsolete - use MX)   |
| MF        | 4 a mail forwarder (Obsolete - use MX)     |
| CNAME     | 5 the canonical name for an alias          |
| SOA       | 6 marks the start of a zone of authority   |
| MB        | 7 a mailbox domain name (EXPERIMENTAL)     |
| MG        | 8 a mail group member (EXPERIMENTAL)       |
| MR        | 9 a mail rename domain name (EXPERIMENTAL) |
| NULL      | 10 a null RR (EXPERIMENTAL)                |
| WKS       | 11 a well known service description        |
| PTR       | 12 a domain name pointer                   |
| HINFO     | 13 host information                        |
| MINFO     | 14 mailbox or mail list information        |
| MX        | 15 mail exchange                           |
| TXT       | 16 text strings                            |

10.28

## Област с подобласт

| [amigo.acad.bg] |                                                    |
|-----------------|----------------------------------------------------|
| acad.bg.        | SOA amigo.acad.bg vdrin.acad.bg. (200310210128800) |
| acad.bg.        | NS server = amigo.acad.bg                          |
| acad.bg.        | NS server = unicom.acad.bg                         |
| acad.bg.        | NS server = ns1.univie.ac.at                       |
| croom8          | A 194.141.0.97                                     |
| croom9          | A 194.141.0.98                                     |
| art             | NS server = amigo.acad.bg                          |
| art             | NS server = unicom.acad.bg                         |
| vtu             | NS server = ns.vtu.acad.bg                         |
| ns.vtu          | A 194.141.4.1                                      |
| vtu             | NS server = amigo.acad.bg                          |
| vtu             | NS server = unicom.acad.bg                         |
| muvar           | NS server = asclep.muvar.acad.bg                   |
| asclep.muvar    | A 212.39.81.180                                    |
| muvar           | NS server = dp20.tu-varna.acad.bg                  |
| dp20.tu-varna   | A 194.141.24.4                                     |
| muvar           | NS server = unicom.acad.bg                         |
| muvar           | NS server = amigo.acad.bg                          |
| gateN           | A 194.141.0.212                                    |
| dis             | A 194.141.0.26                                     |

## Пространство на имената и разрешаване на имената

- върховете на което са разположени имената на компонентите; имената във върховете са на компоненти-директории; обикновено дървото на имената има само един корен
- път в графа на имената – абсолютен (от корена) и относителен път
- графът на имената обикновено е дърво (само с едно входящо ребро за всеки възел – връх, листо) или е ацикличен
- решаване на имената (name resolution) е извличането на идентификатор на компонента при зададено име (и път)
- псевдоним (alias) е допълнително име на компонент:
  - когато графът на имената допуска повече от един път до компонента – пример в UNIX (фиг.10.23)
  - когато съдържанието на възел-лист от графа на имената не е име на компонент а абсолютен път до името на този компонент
- свързване (mounting) на две пространства имена се реализира като възел от едно пространство (mount point) съдържа идентификатор на възел от друго пространство (mounting point)

10.25

## Domain Name System DNS

- DNS е най-голямата разпределна система за имена на компоненти, на която се базира Интернет
- йерархична (т.е. дървовидна) организация на възлите, което позволява ползването на общ етикет за [единственото] входящо ребро и за възела
- етикетите се означават със символни низове без различаване на главни и малки букви до 63B, а с абсолютния път – до 255B
- абсолютният път се отчита от корена и се означава с “.”, която може да се пропусне – courses.FMI.uni-Sofia.bg.
- област (domain) е поддърво в DNS, абсолютният път до нея е името на областта
- съдържанието на възела (т.е. интерпретацията на именованите компоненти) се задава с асоцииран към него списък от ресурсни записи:

10.27

## DNS имплементация

- DNS прилага трислоен модел като поддържа глобалното и административното ниво (локалното ниво е файловата система на възлите)
- зоните се поддържат от [репликирани] сървери на имената
- съответствие: между области и зони
  - когато областта е изградена като една DNS зона, в зоневия файл няма сървери на имената в други зони
  - когато областта съдържа подобласти, които са в отделни зони, зоневия файл съдържа запис с името на подобластта, нейния DNS сървер и неговия адрес (вж. жълтия блок в следващия пример)

10.29

## Итеративно решаване на адресите

- при итеративното решаване на адресите пълното име (с път) – напр. ftp://is.fmi.uni-sofia.bg/t3/rITa1.pdf – се предава на сървера на имената в корена (адресът на чиято реплика е преконфигуриран локално)
- коренът решава обикновено само най-външната област т.е. връща адреса на сървер на имена, който я обслужва (в случая .bg)
- процесът продължава надолу по йерархията, докато се стигне до сървер на имена, който връща адрес на протоколен сървер (адреса на файловата система, поддържаща съответния документ или файл – тук ftp) – фиг. 10.29
- DNS-фазата от решаването на адреса се обслужва при клиента от специален процес – name resolver, а последната стъпка с протоколния обмен се изпълнява от друг клиентски процес

## Рекурсивно решаване на адресите

- при рекурсивно решаване на адресите пълното име – напр. `ftp://is.fmi.uni-sofia.bg/t3/riTa1.pdf` – се предава отново на сървера на имената в корена
- сърверът на имена не връща решения адрес (на следващ сървер) към клиента, а вместо това предава остъгъка от името към този адрес/сървер
- стъпката се повтаря, докато не бъде решен адреса на протоколния сървер, който се връща обратно по йерархичната верига към корена
- решеният адрес се предава към клиентския процес от корена, след което отделен клиентски процес обслужва протоколния обмен с така решения адрес – фиг. 10.30
- предимството на рекурсията е съкращаване на комуникациите (статистически) и по-добра възможност за локално кеширане на адресните решения
- недостатък е централизацията на решаването в сървера на корена –
- затова DNS прилага на глобално ниво итеративния подход, а на административно – рекурсивния

10.32

## Брое на указателите

- асоциира статуса на обекта (компонента) с брояч на указателите (напр. клиентски стъбове) към него със съответното инкрементиране и декрементиране; обект с нулев брояч подлежи на премахване; броячът на указателите се поддържа обикновено от скелетон-стъба на обектния сървер – фиг. 10.32
- при РС този подход (приложен без модификации) поражда проблеми поради комуникационни закъснения и загуби – напр:
  - дублиране на инкрементиращи и декрементиращи съобщения, поради загуба на потвърждения от сървера
  - при наследяване (копиране) на указател към друг клиентски процес инкрементиращото съобщение на новия указател може да закъсне след декрементиращото към 0 съобщение за стария указател
- за преодоляване на комуникационните проблеми се прилага брое на теглото на указателите (weighted reference counting), което преодолява проблема с размножаването на указатели при репликиране на клиентските обекти чрез присвояване на [равна] част от теглото на своя указател на всеки новосъздаден указател
- друг подход е брое на генерациите указатели (generation reference counting), при който освен брояч на поредните указатели се асоциира и с брояч на генерацията: ако напр. клиентски обект от k-генерация създаде n нови обекта (които се явяват k+1 генерация), след което изтрие своя указател, скелетонът в обектния сървер отразява  $G(k) = G(k-1) + G(k+1) = n$ .

10.34

## Недостижими компоненти

- недостижими компоненти (подлежащи на изтриване) са компоненти без път от указатели към тях от някой корен
- те не се засичат по никой от горните методи, а чрез проследяване (tracing-based garbage collection) – проследяване на указателите към всички компоненти (метод с ниска скалируемост!)
- при **унипроцесорите** проследяването се прави по метода mark-and-sweep:
  - с фаза на маркирането на достижимите от корените компоненти и
  - фаза на изчистването, при която системата открива в паметта компоненти, нефиксиращи в маркирания списък, които се изтриват
- вариант: компоненти с открит указател към тях, но преди да е извършено проследяване на техните указатели, се маркират междинно като "сиви" (традиционно "бели" са компоненти, към които не са открити указатели, а "черни" са достижими компоненти, за които проследяването е завършило)

10.36

## Условие за проследяване

- методът mark-and-sweep изисква графа на достижимост да не се променя докато трае оцветяването и изтриването – т.е. спиране на изпълнението на процесите ("stop-the-world"); в разпределен вариант това означава, че всички процеси трябва да синхронизират моментите на стартиране на проследяването и на след това на възстановяване на изпълнението си
- за по-добра скалируемост (вкл. преодоляване на ефектите от "stop-the-world") се прилага проследяване в групи от процеси:
  - процесите се разделят на групи, в които се извършва групово проследяване – асинхронно на останалите групи
  - след като са изчистени всички групи, се извършва глобално проследяване, което се очаква да е по-бързо, тъй като вече са изчистени повечето B-компоненти

## Премахване на неадресираните компоненти

- Garbage collection – в РС обръщението към отдалечени компоненти се базира на локални указатели към тях; отсъствието на такива указатели означава че компонента трябва да се премахне, но наличието им не винаги означава актуалност (напр. циклични указатели между два ненужни компонента)
- при разпределените обекти двойката proxy-skeleton: прокси-стъб обслужва клиентския интерфейс към обекта, а скелетон-стъб – сърверния; обикновено тези две стъб-части обслужват разчистването, защото
- разполагат с информация за текущите обръщания
- могат да маскират тази системна функция от клиентския и сърверния процес
- граф на указателите с множество на корените, които не се премахват дори и когато няма указатели към тях – напр. потребители, системни услуги – фиг. 10.31
- компонентите, които не са пряко или косвено достижими от множеството корени, подлежат на премахване
- поддържането на граф на указателите и на списък с недостижимите компоненти в РС се осъществява с модел на комуникации, съобразен с изисквания за ефективност и скалируемост

10.33

## Списък на указателите

- принципно различен подход за garbage collection е вместо да се броят указателите, скелетонът да регистрира прокси-стъбовете, които извикват обекта, в списък на указателите (reference list) с идемпотентни операции за включване и изключване (мощността на всяко прокси в списъка е 1)
- допълнително предимство на идемпотентността е, че заявките могат да се изпращат няколкократно (напр. за отказоустойчивост) без да се променя резултата в списъка – което не е валидно при броячите
- този метод се прилага в Java RMI – при отдалечено обръщение към обект викация го процес изпраща на скелетона своя идентификатор и след получаване на потвърждение [за включване в списъка указатели] процесът зарежда обектното прокси в адресното си пространство
- ако отдалечен процес P1 предаде копие от обектното прокси на друг п-с P2, P2 изпраща заявка/и за включване в списъка на скелетона и инсталира прокси-стъба след потвърждение
- проблем при горния сценарий: заявка от P1 до скелетона за изключване от списъка преди P2 да заяви включване – ако списъка междувременно стане празен, скелетонът може да изтрие обекта; срещу това се прилага заявка от P1 (също с потвърждение към P1) за предстоящо включване на P2, така че скелетонът поддържа списък на текущите и на предстоящите заявки

10.35



## Mark-and-sweep за разпределени системи

- всеки п-с P<sub>i</sub> стартира собствен колектор, който оцветява прокси- и скелетон-стъбовете, както и самите обекти с Б, Ч и С в следните стъпки:
  - първоначално всички компоненти са оцветени с Б
  - обекти от адресната област на P<sub>i</sub>, които са достижими от P<sub>i</sub> (явяващ се локален корен), се оцветяват С, също така се оцветяват и прокси-стъбовете, заредени от този обект; което означава че техните разпределени обекти са също С
  - до скелетоните съответстващи на "сивите" прокси-стъбове се изпраща съобщение, което оцветява С самите скелетони и техните обекти (скелетоните и техните обекти са отдалечени по отношение на оцветяващия колектор на P<sub>i</sub>)
  - прокси-стъбовете, заредени от отдалечен обект, оцветен С, също стават С; тогава отдалеченият обект и неговия скелетон-стъб стават Ч и скелетонът връща съобщение на адресиращите го прокси-стъбове
  - прокси-стъбовете, получили това обратно съобщение се оцветяват Ч
  - колекторите продължават рекурсивно до завършване на оцветяването т.е. до оцветяване с Б и Ч (накрая няма С-компонети няма)
  - втората фаза е премахване на всички Б-компоненти: обекти, скелетони и прокси-стъбове, (заредени от Б-обектите или асоциирани с тях)

11.1

## 11. Системни средства за реално време

Васил Георгиев

 [ci.fmi.uni-sofia.bg/](mailto:ci.fmi.uni-sofia.bg)  
 [v.georgiev@fmi.uni-sofia.bg](mailto:v.georgiev@fmi.uni-sofia.bg)

## Съдържание

- Синхронизация и системно време
- Протоколи за подреждане
- Глобален статус
- Взаимно изключване
- Разпределени транзакции

## Синхронизиращи алгоритми за системното време

- базират се алтернативно на:
  - времеви **сървер**, който се синхронизира по UTC, или усреднява системното време на възлите
  - **разпределени схеми** за ресинхронизация от тип р2р
- централизирана (сърверна) синхронизация:
  - алгоритъм на Christian (1989 – **пасивен сървер с UTC**): периодични заявки от системните възли към времевия сървер; проблеми:
    - закъснение в цикъла заявка-обслужване-отговор – затова корекцията се прави като към полученото време от сървера се добавя обикновено половината (възможни вариации и по-сложни алгоритми) от закъснението на отговора (измерено на локалната машина) – фиг. 11.4.1.
    - коригира само избързането (винаги!) – налага се постепенни корекции при всяка следваща заявка – напр. корекцията с 2ms вместо установените 10ms (независимо от посоката)
  - алгоритъм на Berkeley UNIX (1989 – **активен сървер**, демон): периодична проверка на локалните системни времена във възлите и изравняване към средна стойност (**без връзка с UTC** предавател) – фиг. 11.4.2.

## Синхронизация за подреждане

- прилага се за подреждане на събития, когато
  - не е важно съответствието между машинното и физическото време – логически часовник
  - не е необходима синхронизация на машинното време между възлите, а само еднакво подреждане на отдалечени събития
- базира се на
  - релация за предходност (happens-before rel.) между събитията:  $a \rightarrow b$  ( $a$  предхожда  $b$ ), транзитивна
  - релация за конкурентност – когато не може да се определи реда на [две] отдалечени събития – напр. в два асинхронни процеса
  - логическо време  $C: a \rightarrow b \Leftrightarrow C(a) < C(b)$ ; то се коригира само в посока нарастване
- в РС логическото време е локално за всеки възел

## Протокол за тотално подреждане

- прилага логическа синхронизация с времеви марки за еднакво подреждане на събитията (получаване на съобщения) при групово предаване (multicasting) – напр. при коригиране на записите в репликирана база данни
- при групово предаване на съобщения с времеви марки изпращащия процес като член на групата получава своите съобщения и то в реда на изпращането им и без загуби
- всеки приемач процес записва получените съобщения в локален буфер по реда на времевите марки и потвърждава приемането до процесите в групата; потвържденията също се маркират (дистанцирано от съотв. съобщение)
- същевременно се прилага и алгоритъм на Лампорт за положителни корекции на локалното логическо време
- всички съобщения – вкл. потвържденията! – са групови (независимо дали са предназначени за всички процеси в групата)
- локалните буфери са опашки (FCFS) от които съобщенията се предават към съответните локални приложения, като се изтриват от буфера (както и техните потвърждения)
- резултат: всички локални буфери са с еднакво подреждане на съобщенията и потока от съобщения към всяко локално приложение е идентичен (N.B.: еднаквото подреждане обаче не гарантира запазване на реда на възникване на събитията в реално астрономическо време  $\Rightarrow$  алгоритъм на Лампорт е приложим за събития, между които няма причинно-следствена връзка – causality)

## Системно време и таймери

- синхронизацията е необходима при:
  - комуникации между процесите
  - подреждане на разпределени събития – право на достъп, бюлетин, транзакции
  - използване на системното време като аргумент – пример take команда в UNIX
- в РС (за разлика от уни- и мультипроцесорите) програмните компоненти може да са разположени на компютри с разлика в системните времена – фиг. 11.3.1 – десинхронизация (clock skew) поради разлика в тактовата честота на осцилаторите и при настройката на системата
- системното време се отчита от таймер – кристален осцилатор + брояч + регистър за броя импулси за 1 сек. – с генерация на системно прекъсване (обикновено с интервал 1 сек.); системният часовник е процес, който отброява прекъсванията  $C$  по таймер
- за глобална координация се използва UTC – Universal Time, Coordinated – което се разпространява чрез късовълнови радиостанции от националните институти по стандартизация и геостационарни сателити
- целта е  $dC/dt = 1, \forall t$ ; реалните осцилатори в масовите компютри работят с относителна грешка  $\rho \approx 10^{-5}$ , т.е.
 
$$1-\rho \leq dC/dt \leq 1+\rho,$$

$$\rho \text{ е максимално отклонение (maximum drift rate) с възможно избързване или изоставане – фиг. 11.3.2}$$
- отклонението между два системни часовника за време  $\Delta t$  е
 
$$\delta \leq 2\rho\Delta t,$$
 и ако това е необходимата горна граница на десинхронизация (skewing), се налага ресинхронизиране с период  $\delta/2\rho$  сек.

## р2р синхронизация

- базира се на периодично общодостъпно предаване на локалното време от всеки възел
- след определено изчакване в началото на всеки период, възлите изчисляват локално време – примерно чрез усредняване с евентуално игнориране на екстремните стойности;
- параметри: период на гласуване  $R$ , период на изчакване  $S \ll R$  и брой на игнорираните екстремни стойности  $m$  (алгоритъмът изисква начален синхронен момент за отчитане на периодите  $T_0$ )
- протокол за мрежово време ([Network Time Protocol](#), NTP; Mills, 1992) – осигурява синхронизация в Интернет с точност до 50 msec.

## Синхронизация с времеви марки (timestamps)

- (Lamport, 1978): синхронизиращи съобщения между възлите с времеви марки на локалните логически времена
  - ако получаващия процес има по-малка стойност на локалното логическо време от марката на изпратеното съобщение, той коригира своя логически часовник (само в положителна посока!) към стойност (марка+1) – фиг. 11.7.
  - изискване: няма две събития с еднакво  $C$  – ако синхронизиращия процес изпраща/приема едновременно две съобщения с времеви марки, той ги дистанцира логически на 1 такт
- допълнително прецизиране на логическото време (за уникални марки) се постига като към целочислената марка се добави процесния идентификатор (или негова производна) като дробна част

## Протокол за съхранено подреждане

- позволява тотално подреждане на събития при запазване на реда им в реално време – напр. при публикуване на дискуссионни и новинарски бюлетини, където е важна не само идентична подредба, но и запазване на причинно-следствената връзка – т.е. съхранено подреждане (causally ordering)
- прилага векторна маркировка (vector timestamp):
  - всеки процес  $P_i$  поддържа свой вектор от броячи  $V_i$ , които елементи отразяват броя събития, настъпили в процесите с съответен индекс –  $V_i[j]$  = брой настъпили събития в  $P_j$ ;  $V_i[i]$  = брой събития в  $P_i$
  - за целта когато  $P_i$  изпраща съобщението  $m$ , към него добавя (т.нар. piggybacking) и текущата стойност на своята вектор  $V_i$  като векторна марка  $v_t$
  - по този начин получаващият съобщението  $m$  процес  $P_j$  е информиран за броя събития, възникнали във всички процеси преди  $P_i$  да изпрати  $m$  – т.е. общия брой събития, от които изпращането на  $m$  може (потенциално) да е следствие
  - при получаването на  $m$   $P_j$  прави корекциите  $V_j[k] = \max\{V_j[k], v_t[k]\}$  и  $V_j[j]++$ , при което  $P_j$  вече разполага с броя събития-съобщения, които предхождат (евентуално като причина)  $m$  (и съответно – ако има такива – може да ги изчака)



## Примерен сценарий за съхранено подреждане

- електронен бюлетин с участие на процесите  $P_i, P_j, P_k$  (и други):
  - $P_i$  – публикува [в групата] статия (съобщение)  $a$ ; при груповото предаване  $P_i$  добавя към  $a$  и векторната марка  $vt(a)=V_i$
  - $P_j$  – публикува пасивно  $a$  след което публикува [в групата] реакцията (съобщение)  $r$ ; при получаването на  $a$   $P_j$  коригира  $V_j$  така че  $V_j[i] > vt(a)[i]=V_i$ ; при изпращането на реакцията  $P_j$  добавя към  $r$  векторната марка  $vt(r)=V_j$ ; (подреждането на събитията се регистрира чрез отношението  $vt(r)[i] > vt(a)[i]$ )
  - $P_k$  – публикува пасивно  $a$  и  $r$ ;  $P_k$  получава двете съобщения (незадължително в коректна последователност) но публикува  $r$  само след като:
    - $vt(r)[i] = V_k[i]+1$  (т.е.  $r$  е точно следващото съобщение, което  $P_k$  очаква от  $P_j$ ) и
    - $vt(r)[i] \leq V_k[j]+1, \forall i \neq j$  (т.е.  $P_j$  не е получил съобщения, които  $P_k$  не е получил към момента на изпращане на  $r$ ; в конкретния пример това е важно само за съобщението  $a$ )

## Алгоритъм за заснемане на глобален статус

- РС се разглежда като набор процеси, свързани с преки симплексни канали (еднопосочни, за разлика от дуплексните и полу-дуплексните) от тип точка-точка (напр. TCP връзки)
- алгоритъмът се иницира от произволен процес  $P$  с регистриране на локалния си статус и изпращане на маркер-заявка за заснемане на глобалния статус по всичките си изходящи канали
- процесът  $Q$  получава заявката по свой входящ канал  $C$  след което
  - [заснемане на процес:] регистрира своя локален статус и размножава заявката по своите изходящи канали;  $Q$  е наследник, а изпращащата заявка процес е предшественик ИЛИ
  - [заснемане на канал:] ако вече е получил заявката (по друг свой входящ канал) и е регистрирал локалния си статус, той регистрира статуса на канала  $C$  – т.е. съобщенията, които е получил по този канал в интервала от регистриране на локалния статус до получаване на последния маркер по канала  $C$
- края на заснемането за  $Q$  е когато получи маркер по всеки свой входящ канал и изпълни горната стъпка; заснетият от него локален статус се изпраща на  $P$  (възможни варианти с цел рекурсивно описание на процеса)
- $P$  разполага с глобалния статус на системата когато получи локалните статуси на своите наследници (и рекурсивно – на техните наследници)
- няколко заснемания могат да бъдат инициирани така че да протичат едновременно – за целта маркерите съдържат идентификатор на инициатора (който се използва и за изпращане на локалния статус)

## Разпределено взаимно изключване (Ricart, Argawala - 1981)

- базира се тотално подреждане на събитията с надеждни (потвърдени) групови комуникации
- заявителят изпраща съобщение с името на критичната зона, своя ид. и локалното време
- всеки получател извършва алтернативно следното
  - връща ОК съобщение ако не е или не чака достъп в тази критична зона
  - ако е в критичната зона, не отговаря, а буферира локално заявката
  - ако е изпратил собствена заявка за същата критична зона, сравнява двете времеви марки и ако има по-късна (по-голяма) марка, изпраща ОК на заявителя, в противен случай не отговаря, а буферира локално отдалечената заявка
- заявителят изчаква ОК от всички останали процеси и заема критичната зона
- след напускане на критичната зона, процесът изпраща ОК на всички заявителя от локалната си опашка за тази зона и ги изтрива от нея
- пример – фиг. 11.14.

## Разпределени транзакции

- транзакциите са механизъм за синхронизация на съвместната работа на устройствата в системата (първоначално при унипроцесорите), на взаимодействащи процеси и др.
- функционират на **принципа “всичко-или-нищо”**: или се изпълняват докрай, или процесите се връщат в състоянието преди началото на изпълнение на транзакцията (примери: обслужване с банкомат, електронна търговия, он-лайн резервации)
- синхронизацията с транзакции се базира на специални **примитиви**, които се поддържат от ОС или се интерпретират като езиково разширение – т.е. обръщения към системата, библиотечни процедури или езикови изрази (специализирани, но в тялото на транзакцията може да присъстват и изрази с общо предназначение)
- наборът транзакционни примитиви е контекстноориентиран, но за синхронизация на обслужването винаги включва `begin_transaction`, `end_transaction`, `abort_transaction` и евентуално `read` и `write` – фиг. 11.16.

## Представяне на глобалния статус

- **глобалния статус** се състои от
  - **локалния статус** на всеки процес
  - **съобщенията в транзит** (напуснали локалния изходен буфер на изпращащата процес, но недоставени в локалния входящ буфер на приемащата процес/и)
- локалният статус на процесите е контекстно-зависим – при разпределена БД той може да включва само записите в БД без междинните резултати на обработка; при mark-and-sweep разчистване може да се състои само от маркировката на скелетоните, прокси и обектите от адресното пространство на съответния процес
- глобалния статус на РС се използва най-често за детекция на мъртва точка (deadlock) или край на разпределената обработка (и в двата случая изпълнението на всички локални процеси е преустановено и няма съобщения в транзит; интерпретацията е предмет на анализ)
- при РС е важна изискването за **свързаност (консистентност)** на глобалния статус – т. нар. **заснемане на РС (distributed snapshot)** – напр. ако п-с  $P$  е получил съобщение от  $Q$ , заснемането трябва да съдържа и запис, че  $Q$  е изпратил това съобщение
- консистентността на заснемането се представя с разрез (cut) – фиг. 11.11.

## Критични зони с взаимно изключване

- в унипроцесорите критичните зони за взаимно изключване на достъпа до споделени ресурси се управлява с механизмите на ключалки-семафори и монитори
- в РС тези подходи се имплементират от централизиран алгоритъм за управление на достъпа, но се прилагат също и разпределени и резервационни алгоритми
- централизирано взаимно изключване
  - базира се на излъчен координатор, към който се отправят заявките за достъп до критична зона
  - заявките се потвърждават по реда на постъпване
  - процесите с непотвърдени заявки изчакват
  - след освобождаване на критичната зона чакащия (блокиран) заявител получава потвърждение (и достъп) – фиг. 11.13.
  - ограничен служебен обмен, но ниска отказоустойчивост; в този вариант заявителя не може да различи изчакване от блокирал координатор

## Резервирано взаимно изключване Token Ring

- базира се на логическо подреждане на п-сите в пръстен; стартиращия процес освобождава съобщението token
- служебното съобщение се предава последователно между процесите, давайки право на текущия процес на достъп до критичната зона, след излизане от която съобщението-token се предава към следващия процес в пръстена
- получаването на token дава права на еднократен достъп в една от критичните зони
- при загубен token възстановяването е контекстнозависимо, тъй като е базирано на времеинтервали
- сравнение между централизираните, разпределените и резервационните алгоритми за взаимно изключване – фиг. 11.15.

## Свойства на транзакциите (ACID), блокови транзакции

- атомарност (Atomic) – т.е. прозрачност – резултата от транзакцията е или като от еднократна моментална операция, или изобщо отсъства все едно не е правен опит да се изпълни (“all-or-nothing”) – напр. транзактно добавяне на байтове към файл преди края на транзакцията файла е достъпен само в началния си вид (без междинни състояния)
- логичност (Consistent) – съхраняване на системните константи – примера с банковия трансфер със запазване на общата сума пари – по време на изпълнение на самата транзакция принципа се нарушава, но друг п-с няма достъп до манипулираната информация, така че нарушението е прозрачно
- изолация (Isolated | serializable) – конкурентните (едновременни) транзакции се изпълняват като последователни съгл. определени принципи на подреждане
- устойчивост (Durable) – след изпълнението на транзакцията резултатите от нея не могат да се отменят
- ACID-| flat- (т.е. блокови) транзакциите не допускат съхраняване и достъп до междинни резултати, което не винаги е желателно, напр. резервацията на серия полети

## Вложени транзакции

- вложени (nested) транзакции – представляват йерархичен дървовиден набор от субтранзакции, първата от които иницира няколко от следващото ниво и т.н. – в съответствие с логическото и каузално (причинно-следствено) разделение на цялата "супертранзакция"; всяка от субтранзакциите е логически независима от изпълнението на останалите (примера с последователните полети – фиг. 11.18.)
- целта е да се постигне ускорено изпълнение при паралелно изпълнение от няколко сървера, но може да се ползват и за съхраняването на междинни резултати
- наборът субтранзакции се счита за изпълнен, само ако главната субтранзакция е изпълнена, а ако не е – заличават се и резултатите на успешно изпълнените дъщерни субтранзакции (което може да породи проблем особено при изпълнение в РС)
- изпълнението на вложените транзакции е рекурсивно: когато главната субтранзакция е изпълнена, за изпълнени се считат и другите завършили субтранзакции по йерархията; резултатите от неизпълнените субтранзакции се заличават

11.20

## Имплементация на транзакциите

- с резервирано работно пространство или с дневник (log-файл)
- резервираното работно пространство изисква при стартирането на транзакцията целият контекст заедно с входно-изходните файлове се разполага в резервирано (private) работно пространство; операциите не се регистрират във файловата система до приключването ѝ
- за оптимизиране, в работното пространство се копират само съответните блокове от файловете, отваряни за четене – както и системния индекс на съответния файл
- обработката се извършва върху копието на блоковете и индекса; след приключване на транзакцията, индекса и блоковете се коригират и във файловата система – фиг. 11.20.
- при метода с log-файл всеки от записите на транзакцията се извършва направо върху блоковете на файловата система, но предварително се регистрира с индекс на блока, старо и ново съдържание (writehead log)
- в случай че транзакцията бъде отменена, регистрационният (log-) файл се използва за възстановяване в обратен ред на записите (LIFO) – "rollback"
- тези методи са приложими и за разпределените транзакции, тъй като субтранзакциите оперират локално

11.22

## Серийно планиране на конкурентни транзакции

- серийното планиране запазва резултата от конкурентните транзакции такъв, какъвто би бил при последователното им изпълнение
- пример – фиг. 11.22. – с две коректни и едно некоректно планиране
- коректното планиране разрешава конфликтните операции
- конфликтни операции са тези, които две (или повече) конкурентни транзакции извършват върху общи данни и поне една от тези операции е запис:
  - четене-запис конфликт
  - запис-запис конфликт
- конфликтът се разрешава чрез заключване на данните или чрез подреждане с времеви марки
- прилагат се два планиращи подхода:
  - песимистичен подход:** операциите се синхронизират *преди* изпълнението им т.е. проверяват се за конфликт и ако да – се подреждат преди да бъдат изпълнени
  - оптимистичен подход:** операциите се синхронизират *след* изпълнението им т.е. изпълняват се целите транзакции и ако накрая се установи че е имало конфликтни операции, поне една от транзакциите се отменя (абортира)

11.24

## Варианти на 2PL

- строго (strict) 2PL**, при което всички заключвания на транзакцията се освобождават след приключване на последното от тях (дори и когато съответната транзакция завършва с отмяна); така се избягва възможността от каскадни отмени на транзакции, която възниква ако са били обработени резултати от отменени впоследствие транзакции (достъпни са само резултати на вече изпълнени транзакции)
- блокировка в мъртва точка (deadlock -) при [strict] 2PL настъпва ако две транзакции заявят едновременно две заключвания но в обратен ред
- за избягване на мъртва точка се прилага:
  - служебно подреждане на заявките
  - времеинтервал за откриване на мъртва точка – когато заключването продължи в рамките на този интервал
  - граф на процесите и заключванията за откриване на цикли
- централизирано 2PL**, при което заявките се обработват от централизиран Д, а достъпът на МД до МД е разпределен; вариант: няколко Д си разпределят контрола за достъп до данните (primary 2PL)
- разпределено 2PL**: всеки Д планира достъпа само до локалните данни, но ако данните са репликирани, съответния Д размножава заявката до възлите с реплики

## Разпределени транзакции

- при тях декомпозицията на супертранзакцията в субтранзакции не следва логическото разделение, а се определя от структурата на разпределения контекст – напр. разпределна база данни, върху всеки от дяловете на която оперира отделна субтранзакция
- пример: междубанков трансфер със субтранзакции върху различни бази данни – фиг. 11.19.
- контраст с блоковите транзакции: блокова е напр. транзакция за начисляване на лихва по сметка (в една база данни)

11.21

## Конкурентно изпълнение на транзакциите

- конкурентното (едновременно) изпълнение на няколко транзакции изисква контрол на достъпа до техния контекст – напр. файлове и БД-записи – така че резултатата да е консистентен т.е. такъв като при последователното им изпълнение
- за целта управлението на транзакциите се разслоява йерархично на 3 нива:
  - мениджър транзакции МТ – транслира примитивите на отделните транзакции в заявки за следващото диспечерско ниво (напр. с ид. на транзакцията и [отдалечен] адрес на данните + управляваща информация)
  - диспечер Д – планира реда и момента за извършване на отделните операции от различните транзакции съгласно планиращ алгоритъм (по методите с ключалки и времеви марки)
  - мениджър данни МД – изпълнява четене и запис в устойчивите структури данни прозрачно за планирането на транзакциите
- конкурентно изпълнение в РС (фиг. 11.21.):
  - във всеки възел се стартира двойка от процесите Д и МД, а за всяка транзакция – отделен МТ
  - МТ изпраща генерираните заявки към съответния Д
  - Д може да изпрати планираните от него заявки и към отдалечени МД

11.23

## Песимистично планиране с двуфазно заключване

- тъй като транзакциите са конкурентни, заявките за заключване подлежат на потвърждение (от Д в зависимост от изискванията на безконфликтното серийно планиране)
- при двуфазното заключване (two-phase locking, 2PL) заключването се разделя на две фази:
  - нарастване (growing phase): процесите на транзакциите заявяват заключване на съответните данни (чрез заявка от съотв. МТ до Д); заключване е необходимо и при четене
  - свиване (shrinking phase): процесите на транзакциите заявяват отключване на съответните данни чрез заявка от съотв. МД до Д – фиг. 11.23.
- важат следните правила за диспечеризация на конкурентните заявки:
  - при заявка за операция, Д проверява конфликтността с вече потвърдените заявки и потвърждава заключването или отлага заявката както и изпълнението на заявяващата транзакция (песимистично планиране)
  - Д освобождава заключване само след като получи потвърждение от ДМ, че операцията е завършила
  - след освобождаване на заключване по заявка на даден МТ (и респ. транзакция), Д не допуска нова заявка от същата транзакция – независимо дали е за същия или друг обект; нови заключвания се допускат преди да е освободено първото от тях; противното е програмна грешка, която отменя самата транзакция

11.25

## Песимистично планиране с времеви марки

- при този метод се маркират както заявките, така и данните
- заявките се маркират с времева марка  $s$  за началото на съответната транзакция  $T$  като се прилага алгоритъма на Лампорт за уникалност на маркиите – т.е.  $s(T)$
- обектите данни  $x$  се маркират с марки за четене  $I$  и запис  $sr(x)$  – съответстващи на транзакционните марки  $s(T_1)$  и  $s(T_n)$  на процесите, които последни са извършили съответните операции
- при конфликт на две заявки се потвърждава тази с по-малка марка (по-ранно стартиране)
- при заявка  $read(T, x)$ :  $s(T) < sr(x) \rightarrow T$  се отменя (абортира) –  $x$  е променен след старта на  $T$
- при заявка  $read(T, x)$ :  $s(T) > sr(x) \rightarrow$  заявката на  $T$  се потвърждава, като  $sr(x) = \max\{s(T), sr(x)\}$
- при заявка  $write(T, x)$ :  $s(T) < sr(x) \rightarrow T$  се отменя (абортира) –  $x$  е прочетен след старта на  $T$
- при заявка  $write(T, x)$ :  $s(T) > sr(x) \rightarrow$  заявката на  $T$  се потвърждава, като  $sw(x) = \max\{s(T), sw(x)\}$
- примери – фиг. 11.25
- планирането с времеви марки води по-често до отмяна на транзакции от това със заключване, защото отменя транзакции, които при заключването само биха били отложени; същевременно при времево маркиране не възниква мъртва точка (поради уникалността и маркиривката на данните)
- варианти: консервативно планиране с времеви марки [Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann 1993.] и многовариантно планиране с времеви марки [Ozsu and P. Valduriez. Principles of Distributed Database Systems. Prentice Hall, 1999.]

## Оптимистично планиране с времеви марки

- конкурентните транзакции се изпълняват докрай без заключване и сравняване на времеви марки, като същевременно се регистрират всички обекти данни, върху които е изпълнено четене или запис
- в края на транзакцията се проверява дали нейните операции са консистентни на операциите на останалите конкурентни транзакции и при откриване на промяна в даден обект след стартирането на тази транзакция, тя се отменя (аналогия с песимистичното времево планиране)
- това планиране се имплементира с резервирано работно пространство за всяка транзакция, чието съдържание се записва във файловата система само при успешно изпълнение на транзакцията
- особености на оптимистичното планиране: висок паралелизъм – няма отлагане и мъртви точки – но при отмяна на транзакция, тя се рестартира отначало
  - при високо натоварване на РС ( $\rho > 80\%$ ) производителността е по-лоша от тази на песимистичното планиране
  - рядко се прилага за РС и понеже се възприема като по-сложно за имплементация