3. Паралелна обработка

Васил Георгиев

 ci.fmi.uni-sofia.bg/

 v.georgiev@fmi.uni-sofia.bg

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

- 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 последоватерно и независимо от изпълнението но други инструкции
- при изпълнение на програмите в ***среди 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 ***среди 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 подпрогроми-процеси се пораждат 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 зависимостие 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 дъга 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

- 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. упражнение 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 анализирания 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

- 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 прелагат **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 четата едновремнно 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 алгоритмиза 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 двоика 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 пораждат цял клас от ПА, базирани на един последователен алгоритъм

Фази на проектирането на ПА

- 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

- 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 (скалируемаст)
  - технологично оптимизиране (напр. намаляване на разходите за кодиране на заданията)

… фази на проектирането на ПА

- 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 настрока на комуникационните канали напр. в системи с комутируеми канали, така че от алгоритъма се изисква да специфицира и комуникациония граф на системата за обработка **5.1**

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 пярялелно при обработката на алгоритъма – това е архитектурнонезависима величина; при размер на проблема W не повече от P(W) процесора могат де се ползват ефективно; съществено е съотношението между паралелните и последователните сегменти на ПА

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

$$R(f) = [f/R_h + 1\text{-}f)/R_l]^{-1}$$

- следователно $f \to 1$ $R(f) \to R_h$ и при $f \to 0$ $R(f) \to R_l$
- **N.B.:** макар че е формулиран за темпове на обработка, закона е в сила и се прилага за агрегирана степен на паралелизма на заданието

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)
  - ***немонтонно*** $S_{p1} > S_{p2}$ за $p2 > p1$ – 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

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 2lb$p$ (4.5.1)
- $S_p = n/[n/p + 2\text{lb}p] = np/[n + 2p\text{lb}p]$; $E_p = n/[n + 2p\text{lb}p]$
- 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 **Tp** единици време **(N.B.** единица време е времето за изпълнение на една елементарна операция**)** е

$$C_p = pT_p$$

- т.е. **Cp** 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 Op на брой операции с **p** процесора е

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

- т.е. **Up e** отношението на действителните към потенциалните операции при обработка на съответния ПА

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 *Op* на брой операции при обработка с **p** процесора е

$$R_p = O_p / O_1 > 1$$

(където O_1 е броя операции при обработка на уникомпютър) т.е. R_p е критерий за свръхтовара, който се поражда от паралеланата обработка на алгоритъма; p и n (размера на проблема) са аргументи на R_p но – в зависимост от класа ПА – участват с различна тежест

Цена и коефициент на използване

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

$$C_p = pT_p$$

- т.е. **Cp** 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 Op на брой операции с **p** процесора е

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

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

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 който 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

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 не-йерархични модели **(P2P - Jxta)**

Конвенционален псевдокод за паралелни алгоритми

- псевдокодът (както и езиците 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>
```

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 парлелна 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)

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

- 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
```

Синхронизационни конвенции, семафори

- синхронизационните схеми биват
  - контрол на достъп – семафори и монитори
  - контрол за последователност – бариери
- променлива от тип семафор 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

Синхронизиращ псевдокод със семафор

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

Синхронизация с монитори

- мониторите са разширение на семафорите, 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;
    Resource_is_free: Condition;
Procedure Get_resource
  begin
    is (Resource_is_free) then
      wait(Resource_is_free)
    Resource_in_use = true
  end
...
Procedure Release_resource
  begin
    Resource_in_use = false
    signal(Resource_is_free)
end
end Monitor
```

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:

Псевдокод без синхронизация

```
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])
  }
```

Псевдокод с бариерна синхронизация

```
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])
  }
```

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

- 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 грид – пропорцинално натоварване на ресурси с различна собственост и администрация
- източници на дисбаланс
  - нерегулярност на пробема 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

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 свръхтовар)
  - генетични и Монте Карло алгоритми – свързани са с генериране на възможни варианти на декомпозицията и оценяването им, така че да се избере оптималния

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 ассоцииране на управляващите функции с всеки от обработващите процеси

Централизирано динамично балансиране

- 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**)
- пулът 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

Разпределено динамична 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 двоичнор тъй като разделянето на [под-]проблема на две очаквано равно части е по-лесно за алгоримиране и за прилагане на рекурсия

P2P динамично балансиране

- то е форма на по-пълно 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 послеващ балансиращ трансфер на генерираните подзадания межу 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)

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!) “съседи”
- в горния случай като средство за повишаване на линейността на лгоритъма се избягват схеми когато всички възли са “съседни”; вместо това се формират виртуални топологични структури – линия, пръстен, хиперкуб и др. (обикн. нейерархични) топологии; когато валентността на процесите е по-голяма от 1, може да се прилага циклично или случайно тъсене 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

Системи за динамично балансиране

- информационна, локационна и трансферна стратегия
  - функции
  - граф
  - разпределение

клъстерно, мултиклъстерно и c2c планиране

- синхронно балансиране – co-scheduling
  - Koala (http://www.omii.ac.uk/repository/project.jhtml?pid=122, & http://www.cs.vu.nl/~kielmann/asci-a14/slides/koala/koala.pdf)
- асинхронно балансиране – htc (High troughput computing), volunteer computing
  - Condor/Condor-G (http://www.cs.wisc.edu/condor/), Boinc (http://boinc.berkeley.edu/)
  - балансират се нископриоритетните процеси на опортюнистичните потребители –във фонов режим (background priority)

Еталонни паралелни алгоритми

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