**3. Паралелна обработка**

**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 й с еднакви данни е винаги един и същ. Изпълнението на всяка инструкция е последоватерно и независимо от изпълнението на 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 Multiplе 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 PМD); при този подход отделните подпрограми-процеси се пораждат като дъщерни на един главен процес.

**Граф на процесите**

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. 3.5):

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) – 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

**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)

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 (3.11):

1) 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.

2) комуникации(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.

3) 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 х-ки:

- 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)

4) разпределяне (mapping) – незадължителна фаза, 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

**Метрика и анализ на производителноста**

- 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 Р – 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 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 *Rh* и ниско-паралелна *Rl*, които са в съотношение *f*:(1-*f*) по брой на генерирани резултати (междинни и крайни) – общата интензивност на обработка е

*R*(*f*) = [*f*/*Rh* + 1-*f*)/*Rl*]-1. Следователно при

*f* → 1, *R*(*f*) → *Rh* и *f* → 0 *R*(*f*) → *Rl* 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 3.16**

- При оценка или измерване на ускорението (*Sp* = *T*1/*Tp* ) се приема, че всички процесори са с идентична производителност; поради наличие на комуникационни и синхрониз. закъснения: 1 < *Sp* < *p*

Аномалии:

***суперлинейно*** *Sp* > *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.

***немонтонно*** *Sp1* > *Sp2* за *p2* > *p:* често срещана аномалия.

**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 *Cp* = *pTp*

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 Ор на брой операции с p процесора е

*Up* = *Op*/*Cp* = *Op*/(*pTp*)

т.е. Up e отношението на действителните към

потенциалните операции при обработка на съответния ПА

**Алгоритмична сложност**

Коректността на даден ПА е архитектурно-независима, 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 *О* оценява времевите и пространствени характеристики на обработка – времевата сложност *Т* е се задава в брой елементарни операции и комуникации (от който се получава времето за обработка в

дадена архитектура), а пространствената сложност *М* в брой алокирани регистри и клетки памет (т.е. *О* = *О*(*Т*, *М*));

оценкта се дава обикновено като долна и горна граница на тези величини или с приближение – асимптотична сложност

**Конвенционален псевдокод за паралелни**

**алгоритми**

- Псевдокодът (както и езиците за прогр.) e 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, деф. в границите на блока; по един процес се създава за всяка нейна ст. (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 (асинхр. 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);

**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 нeя се извършват операциите:

- установяване на състоянието (активно или пасивно) (wait)

- блокиране на процес (wait)

- възстановяване от блокиране (signal)

- wait(S) e заявка за достъп до критичната зона, която се потвърждава ако S>0 (и S се декрементира); в противан 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

**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

**Задачи на балансирането на изчисл. товар (Load Balancing, Resource Мanagement, 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.

Източници на дисбаланс:

- нерегулярност на пробема при паралелизъм по данни

-недетерминистични алгоритми за обработка, напр. при неизвестен бр. итерации за дости-гане 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 й

- 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