9. Потоково, функционално и SIMD програмиране

Васил Георгиев

 ci.fmi.uni-sofia.bg/

 v.georgiev@fmi.uni-sofia.bg

Съдържание

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

Императивно и потоково (data flow) програмиране

- 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 й 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 изпълнят едновременно (асинхронно) – в зависимост от наличните ресурси на изпълнителната платформа
  - паралелизмът е на инструкционно ниво

Потоково и функционално програмиране

- 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 разглеждата като взаимни инверсии – което резултира в хибридния 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

Потокови изчисления

- 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 обработка 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)

Потокова програма

- потоковата програма е формално описание на обработката като мрежа, която отразява зависимистите между данните, а опреациите които се извършват върху тях са по-скоро като маркировка на възлите – последователността, в която ще се изпълнят 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 транслация от командна към потокова програма и обратно
- в командната програма също може да се укаже явно парарлелизъм (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 потоковито програмиране е сериозен конкурент

Пример на потокова прогарама

- 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
```

Потокови езици

- 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 Uinevrsity of California - Irvin [Arvind, 1978] и VAL – Value Oriented Algorithmic Language на MIT [Ackerman, 1979]

Потокови езици - особености

- 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
- потоковите езици са априкативни – ориентирани са към генериране на стойности, които се използват за изчислевие 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

Дисциплина на възлите в потоковите езици

- работата на възлите се определя от наличието на съответните входни стойности (tokens) – при готовност се стартира предвидената обработка на входните стойности (node firing) и резултатаите се предават към следващи възли по съответните дъги, след което възелът е в престой 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

Интерпретация на потоковите езици

- възлите се представят като структури в паметта; потоковите 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

VAL (Value-oriented Algorithmic Language)

- записите на този език съдържат неявно описание на алгоритмичния паралелизъм и при интерпретация се представят като потоков 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

Съставни типове данни във VAL

- 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:Experssion1; 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:BOLLEAN];
rec[A:1; B:2; C:3.14; D:'y'; E:FALSE]    /* elements' assignment
F := (rec.A = 1)                         /* F = TRUE
```

Изрази и функции във VAL

- дефиниция на функция:

```
function Class (Param: BOOLEAN returns INTEGER);  /* function  definition
                {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 върщаните 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
```

Паралелни изрази във VAL

- паралелното изпълнение на изрази се задава по някой/и от следните способи:
- 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
```

Примери за паралелни изрази във VAL

- 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 55 = 1 + 4 + 9 + 16 + 25):

```
forall Calc in [1, 5]
  eval PLUS Calc * Calc
endall
```

Условия и цикли във VAL

- 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
```

Функционално програмиране

- функционалното програмиране е близко по съдържание и форма до потоковото програмиране и също е средство за специфициране на паралелна интерпретация с представяне на изчисленията в последователна форма – функционалните езици се разглеждат като хибридизация на императивните и потоковите
- поддържа се принципа за еднократно присвояване на променливите, който елиминира преодолимите 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 улесненатa 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.

Езикови принципи на функционално програмиране

- принципите, на които се базират езиците за ФП, са разработени от автора на Фортран John Backus през 70те години на XX 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
- опрерации 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 примитиви

- примитиви за избор са **FIRST**, **LAST** и **TAIL**:

  x1 ← FIRST(x1, x2,…, xn)

  xn ← FIRST(x1, x2,…, xn)

  <x2,…, xn> ← FIRST(x1, x2,…, xn)

- примитивите за структуриране **ROTR**, **ROTL**, **LENGTH** и **CONS** се използват за структурни операции върху елементите:

  <xn, x1,…, xn-1> ← ROTR(x1, x2,…, xn)

  <x2,…, xn, x1> ← ROTL(x1, x2,…, xn)

  n ← LENGTH(x1, x2,…, xn)

  <x, x1,…, xn> ← CONS(x,<x1, x2,…, xn>)

- аритметични бинартни операции: **+**, **-**, *****, **div** и **|** (за остатък):

  residue_x1_by_x2 ← |:<x1, x2>

- предикатни операции със стойност T или F
- логически операции върху булеви аргументи
- операция за идентичност: **x ← ID:x**

Функционални форми

- композиция на две функции: **(f ○ g):X ≡ f:(g:X)**
- конструкция на *n* функции: **[f1, f2,…, fn]:X ≡ <f1:X, f2:X,…, fn:X>** - пример: **[min, max, avg]:<1, 2, 3, 4, 5> = <1, 5, 3>**
- α-обобщение (apply_to_all): α **f:<x1, x2,…, xn> = <f:x1, f:x2,…, f:xn>**
- включване: **/f:<x1, x2,…, xn> = <x1, /f:<x2,…, xn>** - пример: **/+:<1, 2, 3, 4, 5> = 15**

Езици за функционално програмиране

- FP не е развил достатъчно изразни 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 потоковото 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

SISAL

- 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 производен 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

Типове данни в SISAL

- SISAL поддържа скаларните типове цял, реален, символен, булев и двоен, както и структурите масив, запис, обединение (union) и поток (stream)
- масивите еднодименсионни (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

- 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 стъойностите 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 разграничава с префикса old
    - клауза за резултатите – резултатът е крайната стойност на името на съответния цикъл или редукция 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

Примери за паралелни изрази в SISAL

```
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 компилатор

- компилаторът на Сисал Osc е със сложна структура тъй като транслацията от абстрактните спецификации на Сисал до обектен код за съответнта паралелна архитектура се извършва в седем фази
- транслацията от Сисал към междинната форма IF1 се извършва от парсер като резултата е ацикличен потоков граф с функционална семантика: възлите са или елементарни операции (вкл. манипулции на масиви и потоци) или съставни възли, съдържащи подграфи
- свързването 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 транслацията 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

SISAL ядро

- 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 n процесора (където броя процесори се задава с атрибут на командата за стартиране на приложениетои всъщност може да не съответства на актуалния брой процесори в архитектурата) ядрото разделя циклите на n 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 еднопроцисорни архитектури и ускорения между 7.3 и 9.0 върху 10-процесорна SISD архитектура (Cray Y-MP – 1992)

Особености на програмирането за SIMD

- 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 обарботка на данни (ПОД) като правило поражда много по-високо ниво на паралелизма отколкото паралелизма по управление (който обикновено е някаква форма на конвейризация) – дори и когато последния се прилага при същата фина грануларност – на ниво инструкция; по-високия 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];      }
```

Езикови разширения за ПОД

- 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 разпределение

ПОД за MIMD

- 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

ПОД със C*

- C* е език за ПОД със разширен синтаксис на стандартния C и елементи на ООП като в C++, който представя изпълнителната архитектура като интерфейсен или главен (front-end) унипроцесор, разширен с ко-процесори (ПЕ) за SIMD обработка (back-end) – 9.32
- типовете 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 порграмата)
- програмата се състои от последователни блокове за паралелно (domain – върху ПЕ) и последователно изпълнение (само върху главния процесор)
- данните са скаларни (декларират се като mono и се зареждат в паметта на главния процесор) или векторни – poly, които се разпределят в локалните памети на ПЕ
- транслацията 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*

- 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 димензии 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
```

Шаблонни обръщения в C*

- обръщението към елементите на шаблона е с ляво единично индексиране, което съответства на алокацията им в ПЕ: **`[0]array1`** – елемента в първия процесор
- шаблонните 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`

Паралелни операции в C*

- когато поне един от операндите е деклариран като паралелна променлива, операцията се изпълнява паралелно, за което е небходимо:
  - 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
```

Редукции в C*

- C* дефинира набор от вградени оператори (редукции), с които основни операции върху шаблонни паралелни операнди, чийто 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
+=	скаларна сума на елементите на паралелна шаблонна променлива
-=	негативна сума на елементите
&=	побитова конюнкция на елементите
^=	побитово изключващо ИЛИ на елементите
|=	побитова дизюнкция на елементите
>?=	максимална стойност на елементите
<?=	минимална стойност на елементите

- пример:

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

Паралелни операции върху подмножества елементи

- изразът **where** с опция **else** дефинира подмножества от елементите на паралелни структури – "активни позиции" – върху които се извършва обща паралелна опареция:

```
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
```

Комуникации в C*

- обмена 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:

1	2	3
4	5	6
7	8	9

source1:

	1	2
	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 резулатата – 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]]
```

Елементи на ЕПОД FORTRAN90

- 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

Декларации във FORTRAN90

- 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 стъндартния 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

- декларацията на променливи-масиви се прави с атрибута **DIMENSION** , чиито аргументи указват броя димексии 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
INTEGER A(10), B(10), C
DO I = 1, 10, 1
  A(I) = B(I) + C
END DO
```

```
/* FORTRAN90
INTEGER C
INTEGER, DIMENSION () :: A,
B
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

Многодименсионни масиви във FORTRAN90

- 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
```

Вградени (“intrinsic”) функции върху масиви във FORTRAN90

- 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)