Paralelização
Introdução a vetorização, OpenMP e MPI

2 – Vetorização

Paulo Penteado
IAG / USP

pp.penteado@gmail.com

Esta apresentação: http://www.ppenteado.net/ast/pp_para_on/pp_para_on_2.pdf
Arquivos do curso: http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf

Programa

1 – Conceitos

- Motivação
- Formas de paralelização
  - Paralelismo de dados
  - Paralelismo de tarefas
- Principais arquiteturas paralelas atuais
  - Com recursos compartilhados
  - Independentes
- Paradigmas discutidos neste curso:
  - Vetorização
  - OpenMP
  - MPI
- Escolhas de forma de vetorização
- Algumas referências
- Exercícios – testes de software a usar no curso

Slides em http://www.ppenteado.net/ast/pp_para_on_1.pdf

Exemplos em http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf
pp.penteado@gmail.com

Programa

2 – Vetorização

- Motivação
- Arrays – conceitos
- Organização multidimensional
- Arrays – uso básico
- Arrays – row major x column major
- Operações vetoriais
- Vetorização avançada
  - Operações multidimensionais
  - Redimensionamento
  - Buscas
  - Inversão de índices
- Algumas referências
- Exercícios - vetorização

Slides em http://www.ppenteado.net/ast/pp_para_on_2.pdf

Exemplos em http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf
pp.penteado@gmail.com

Programa

3 – OpenMP

- Motivação
- Características
  - Diretrizes
  - Estruturação
- Construções
  - parallel
  - loop
  - section
  - workshare
- Cláusulas
  - Acesso a dados
  - Controle de execução
- Sincronização
  - Condições de corrida
- Exercícios - OpenMP

Slides em http://www.ppenteado.net/ast/pp_para_on_3.pdf

Exemplos em http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf
pp.penteado@gmail.com

Programa

4 – MPI

- Motivação
- Características
- Estruturação
- Formas de comunicação
- Principais funções
  - Controle
  - Informações
  - Comunicação
- Boost.MPI
- Sincronização
  - Deadlocks
- Exercícios - MPI

Slides em http://www.ppenteado.net/ast/pp_para_on_4.pdf

Exemplos em http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf

pp.penteado@gmail.com

Vetorização

A forma mais simples de paralelização de dados

Apesar do nome, se refere a arrays 1D, 2D, 3D ou mais (não só vetores).

Quando operações sobre diferentes elementos do array são independentes (o que é o mais comum), podem ser paralelizadas facilmente.

Todas as unidades (threads) fazem o mesmo trabalho, apenas operando sobre elementos diferentes.

A divisão do trabalho é feita implicitamente:

- **A semântica usada expressa a operação vetorial**
- Cabe ao compilador / interpretador dividir o trabalho entre as unidades e juntar os resultados no final.*
- Limitado a ambientes de memória compartilhada.

Vetorização de código é importante mesmo que não se vá usar paralelização:

- Torna o código mais legível, organizado e robusto.
- Mesmo sem paralelização pode tornar o programa mais rápido.

*Estas mesmas tarefas podem ser paralelizadas explicitamente pelo programador, mesmo em memória distribuída, mas o nome **vetorização** não se refere a estes casos. Exemplos em OpenMP e MPI na próxima aula.

Arrays - definição

Vetorização se refere ao uso de operações sobre **arrays**:

- **O contêiner mais simples***, implementado até em velhas linguagens, e o de uso mais comum em astronomia.
- Um conjunto sequencial de elementos, organizados **regularmente**, em 1D ou mais.
- Já não presente nativamente em algumas novas linguagens dinâmicas (Perl, Python sem Numpy).
- Às vezes chamado de ***array*** só quando tem mais de 1D (MD), e para 1D é chamado de **vetor**.
- Arrays 2D às vezes são chamados de **matrizes**
  - Em algumas linguagens (ex: R, Python+Numpy), ***matrix*** é diferente de arrays genéricos.

*contêiner: uma variável que armazena um conjunto (organizado) de valores. Arrays são só uma dos tipos de contêiner (não o único, e nem o mais importante).

Arrays - características

Homogêneos (todos os elementos são do mesmo **tipo** (real, inteiro, string, etc.))

Estáticos (não é possível mudar o número de elementos)

Sequenciais (elementos armazenados em uma ordem)

Organizados em 1D ou mais (MD).

Acesso aos elementos através de seu(s) índice(s).

São o principal meio de fazer vetorização:

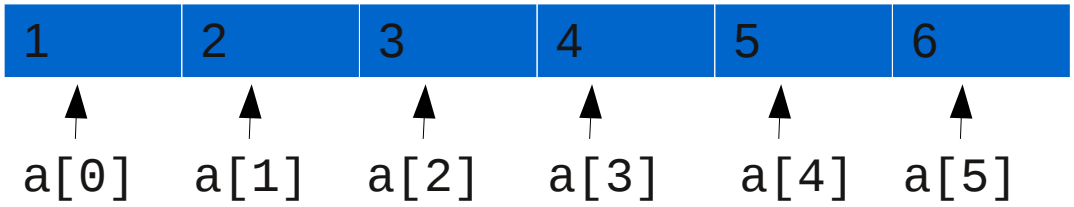
- 1D é muito comum.
- MD é com freqüência desajeitado (2D ocasionalmente não é tão ruim): **IDL e Python+Numpy têm arrays MD de muito alto nível** (adiante).

Internamente, todos os elementos são **armazenados em uma seqüência 1D, mesmo quando há mais de uma dimensão** (memória e arquivos são unidimensionais).

- **Em mais de 1D, são sempre regulares** (cada dimensão tem um número constante de elementos).

Arrays

1D



Ex. (IDL):

```
IDL> a=bindgen(6)+1
```

→ Gera um array de tipo ***byte***, com 6 elementos, com valores 1 a 6.

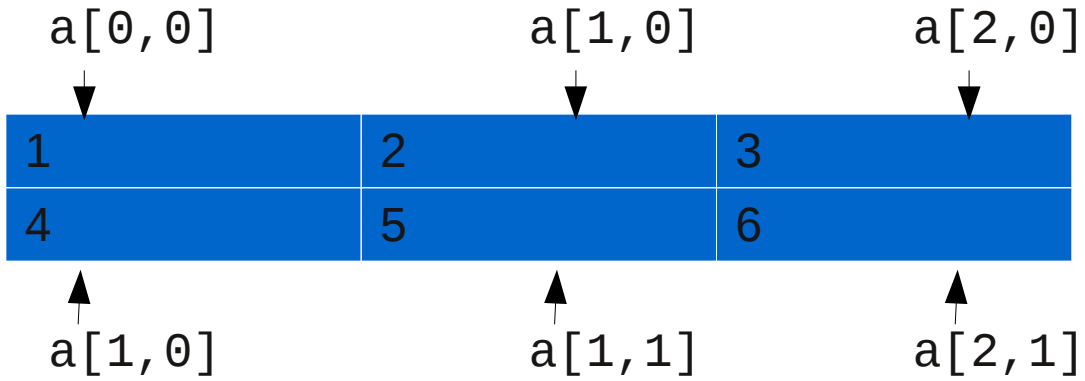
```
IDL> help,a
A               INT       = Array[6]
```

```
IDL> print,a
      1       2       3       4       5       6
```

O mais comum é índices começarem em 0. Em algumas linguagens, pode ser escolhido.

Arrays

2D



Ex. (IDL):

```
IDL> a=bindgen(3,2)+1
```

Gera um array de tipo ***byte***, com 6 elementos, em 3 colunas por 2 linhas, com valores 1 a 6.

```
IDL> help,a
A               INT       = Array[3, 2]

IDL> print,a
       1       2       3
       4       5       6
```

São regulares: não podem ser como

1	2	3	4	5	6			
7	8	9	10					
11	12	13	14	15	16	17	18	19

Arrays

3D costuma ser pensado, graficamente, como uma pilha de “páginas”, cada página sendo uma tabela 2D (ou um paralelepípedo). Ex. (IDL):

```
IDL> a=bindgen(4,3,3)
```

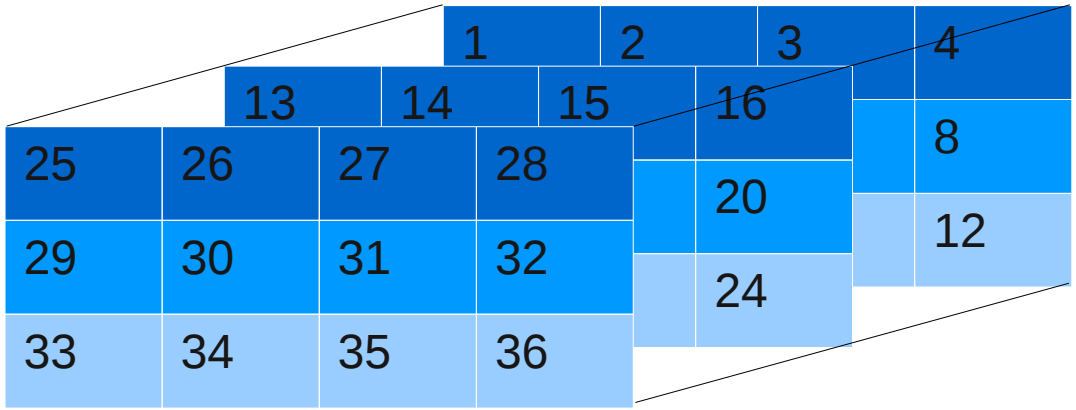
Gera um array de tipo ***byte***, com 36 elementos, em 4 colunas, 3 linhas, 3 “páginas”, com valores 0 a 35.

```
IDL> help,a
A               BYTE      = Array[4, 3, 3]

IDL> print,a
   0   1   2   3
   4   5   6   7
   8   9  10  11

  12  13  14  15
  16  17  18  19
  20  21  22  23

  24  25  26  27
  28  29  30  31
  32  33  34  35
```



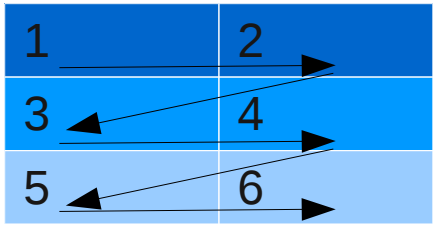
Para mais que 3D, a imagem gráfica costuma ser conjuntos de pilhas 3D (para 4D), conjuntos de 4D (para 5D), etc.

Arrays - armazenamento MD

Se internamente são sempre seqüências 1D, como são armazenados arrays MD?

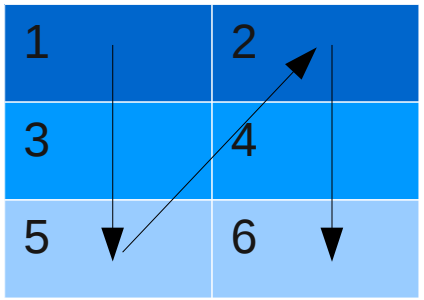
As várias dimensões são varridas ordenadamente. Ex (2D): a[2,3] - 6 elementos:

1)
```
a[0,0] a[1,0] a[2,0] a[0,1] a[1,1] a[2,1]
```
Posição na memória:
```
0      1      2      3      4      5
```



ou

2)
```
a[0,0] a[0,1] a[1,0] a[1,1] a[2,0] a[2,1]
```
Posição na memória:
```
0      1      2      3      4      5
```



Cada linguagem faz sua escolha de como ordenar as dimensões.

Column major - primeira dimensão é contígua (1 acima): IDL, Fortran, R, **Python+Numpy, Boost.MultiArray**
Row major - última dimensão é contígua (2 acima): C, C++, Java, **Python+Numpy, Boost.MultiArray**

Linguagens podem diferir no uso dos termos ***row*** and ***column***.

Graficamente, em geral a dimensão “horizontal” (mostrada ao longo de uma linha) pode ser a primeira ou a última. Normalmente, a dimensão horizontal é a contígua.

Arrays – uso básico

Acesso a elementos individuais, pelos M índices (MD), ou um único (MD ou 1D). Ex. (IDL):

```
IDL> a=dindgen(4)
IDL> b=dindgen(2,3)
IDL> help,a
A               DOUBLE    = Array[4]
IDL> help,b
B               DOUBLE    = Array[2, 3]
IDL> print,a
       0.0000000       1.0000000       2.0000000       3.0000000
IDL> print,b
       0.0000000       1.0000000
       2.0000000       3.0000000
       4.0000000       5.0000000
IDL> print,a[2]
       2.0000000
IDL> print,a[-1]
       3.0000000
IDL> print,a[-2]
       2.0000000
IDL> print,a[n_elements(a)-2]
       2.0000000
IDL> print,b[1,2]
       5.0000000
IDL> print,array_indices(b,5)
           1           2
IDL> print,b[5]
       5.0000000
```

Retornam arrays de **doubles** onde cada elemento tem o valor de seu índice; há versões para os outros tipos numéricos.

Índices negativos são contados a partir do final (Python+Numpy, R, IDL≥8): -1 é o último, -2 é o penúltimo, etc.

Elementos de arrays MD podem ser acessados também por seu índice 1D (IDL, Python+Numpy)

Arrays – uso básico

Acesso a fatias (*slices)*: conjuntos regulares*, 1D ou MD, contíguos ou não. Sintaxe semelhante em IDL, Python+Numpy, R, Fortran, Boot.MultiArray. Ex. (IDL):

```
IDL> b=bindgen(4,5)
IDL> print,b
   0   1   2   3
   4   5   6   7
   8   9  10  11
  12  13  14  15
  16  17  18  19
IDL> c=b[1:2,2:4]
IDL> help,c
C               BYTE      = Array[2, 3]
IDL> print,c
   9  10
  13  14
  17  18
IDL> print,b[*,0:2]
   0   1   2   3
   4   5   6   7
   8   9  10  11
IDL> print,b[1:2,0:-1:2]
   1   2
   9  10
  17  18
IDL> print,b[1,2:0:-1]
   9
   5
   1
```

Elementos das colunas **1 a 2**, das linhas 2 a 4

Todas as colunas, linhas 0 a 2

Colunas **1 a 2**, linhas **0 a última (-1)**, de **2 em 2** linhas (com um passo (*stride*) 2)

O passo pode ser negativo, para andar na ordem reversa

*Em Numpy há fatias não regulares, através da escolha de passos.

Arrays – uso básico - slices

Ex. (Fortran):

```
integer, parameter :: m=5,n=4
integer :: b(n,m),i,j,c(2,3),d(0:n-1,-1:-1+m-1)
foreach (i=1:n,j=1:m) b(i,j)=i+j*n
```

Primeiro índice em 1 por default, mas pode ser escolhido algo diferente

```
   0   1   2   3
   4   5   6   7
   8   9  10  11
  12  13  14  15
  16  17  18  19
```

```
c=b(2:3,3:5)
```

Elementos das colunas **2 a 3**, das linhas 3 a 5

```
   9  10
  13  14
  17  18
```

```
d=b(:,1:3)
```

Todas as colunas, linhas 1 a 3

```
   0   1   2   3
   4   5   6   7
   8   9  10  11
```

```
b(2:3,::2)
```

Colunas **2 a 3**, linhas da **primeira (1) à última (5)**, de **2 em 2** linhas (com um passo (*stride*) 2)

```
b(2,3:1:-1)
```

O passo pode ser negativo, para andar na ordem reversa

Arrays – uso básico

Dimensões de tamanho 1 também podem contar: um array [1,n] não é o mesmo que um array [n]. Ex. (IDL):

```
IDL> e=bindgen(4)
IDL> help,e
E               BYTE      = Array[4]
IDL> print,e
   0   1   2   3
IDL> f=bindgen(1,4)
IDL> help,f
F               BYTE      = Array[1, 4]
IDL> print,f
   0
   1
   2
   3
IDL> g=bindgen(4,1)
IDL> help,g
G               BYTE      = Array[4]
IDL> print,g
   0   1   2   3
```

1D, 4 elementos

2D, 4 elementos: 1 coluna, 4 linhas

2D, 4 elementos: 4 colunas, 1 linha: equivalente a 1D, pois 1D é uma linha

Arrays não são o mesmo que matrizes: matrizes são 2D, e em algumas linguagens (Python+Numpy, R) há subclasses **`matrix`**, mais limitadas, mas otimizadas para álgebra linear. Em Java, há matrizes no pacote **`Jama`**, que provê funcionalidade 2D e álgebra linear.

A principal importância de arrays está em operações vetoriais, (adiante, onde estão os exemplos para usos de arrays).

Arrays - que importa se o array é *row* ou *column major*?

1) **Operações vetoriais** (adiante): para selecionar vários elementos contíguos.

2) **Operação entre linguagens diferentes:**

- Chamando rotinas de outras linguagens, acessando arquivos e conexões de rede escritos / a serem lidos em outras linguagens.

Arrays - que importa se o array é *row* ou *column major*?

3) Semântica de literais. Ex. (IDL):

```
IDL> a=[[1,2,3],[4,5,6]]

IDL> help,a
A               INT       = Array[3, 2]

IDL> print,a
       1       2       3
       4       5       6

IDL> b=[[[1,2,3,4],[5,6,7,8],[9,10,11,12]],[[13,14,15,16],
[17,18,19,20],[21,22,23,24]]]

IDL> help,b
B               INT       = Array[4, 3, 2]

IDL> print,b
       1       2       3       4
       5       6       7       8
       9      10      11      12

      13      14      15      16
      17      18      19      20
      21      22      23      24
```

Arrays - que importa se o array é *row* ou *column major*?

4) Concatenações. Ex. (IDL):

```
IDL> a=[1,2,3]
IDL> b=[4,5,6]
IDL> c=[a,b]
IDL> help,c
C               INT       = Array[6]
IDL> print,c
       1       2       3       4       5       6
IDL> d=[[a],[b]]
IDL> help,d
D               INT       = Array[3, 2]
IDL> print,d
       1       2       3
       4       5       6
IDL> e=[[[a],[b]],[[-d]]]
IDL> help,e
E               INT       = Array[3, 2, 2]
IDL> print,e
       1       2       3
       4       5       6

      -1      -2      -3
      -4      -5      -6
```

Concatenamento na primeira dimensão (a da esquerda)

Concatenamento na segunda dimensão

Concatenamento na terceira dimensão

Concatenações com o próprio array (ex. **`a=[a,b]`**) criam um novo array, e copiam os valores para o novo.
O array (**a**) **não é redimensionado**: arrays são estáticos (não mudam de tamanho).

Arrays - que importa se o array é *row* ou *column major*?

5) Eficiência:

Se o array tem que ser percorrido, é mais eficiente (**especialmente em disco**) o fazer na mesma ordem usada internamente: o acesso é sequencial, sem idas e vindas.

Ex: para percorrer todos os elementos do array *column major*

a[0,0] (a[0]) : 1	a[1,0] (a[1]) : 2
a[0,1] (a[2]) : 3	a[1,1] (a[3]) : 4
a[0,2] (a[4]) : 5	a[1,2] (a[5]) : 6

Na mesma ordem de armazenamento, é uma passagem direta pela memória (ex. IDL)

```
for j=0,2 do begin
  for i=0,1 do begin
    k=i+j*2
    print,i,j,k,a[i,j]
    do_some_stuff,a[i,j]
  endfor
endfor
```

```
i       j       k       a[i,j]
 0       0       0          1
 1       0       1          2
 0       1       2          3
 1       1       3          4
 0       2       4          5
 1       2       5          6
```

Sem idas e voltas (mostrado pela variável **k**, que indica que posição na memória foi usada).

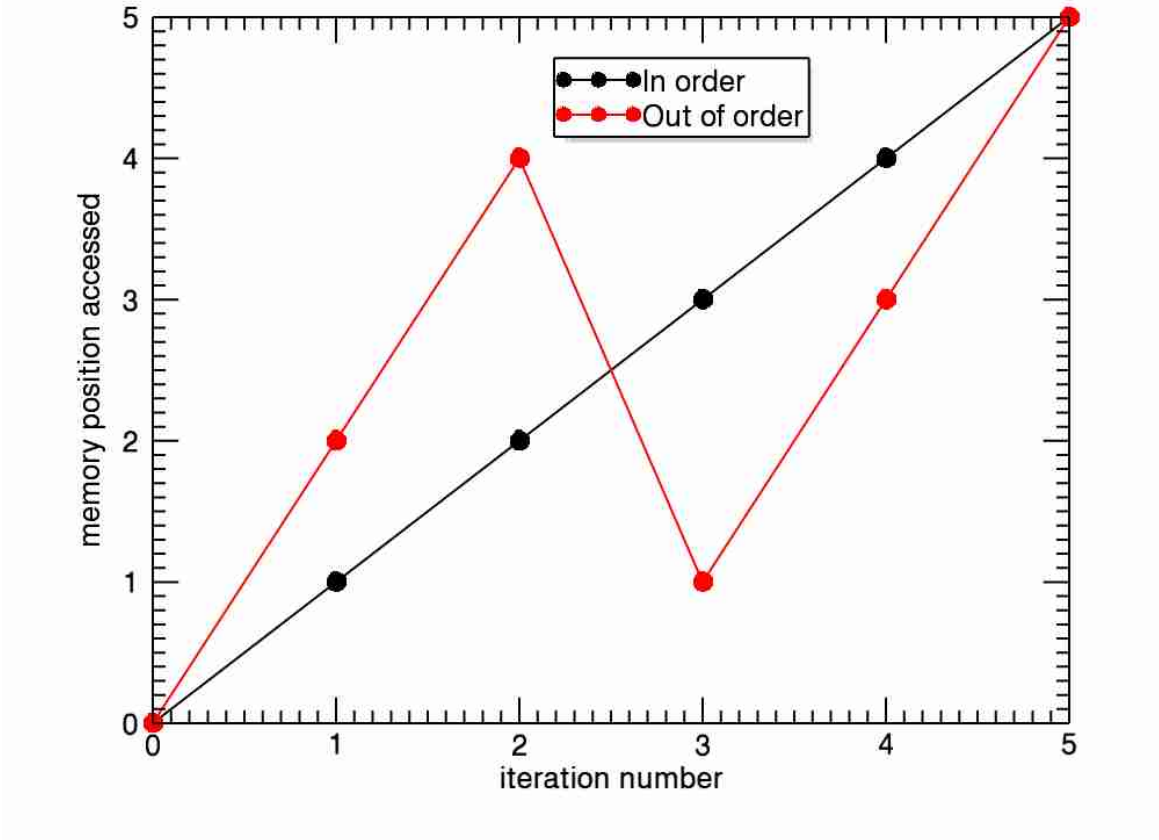
Arrays - armazenamento MD - exemplos

Já se os elementos são lidos fora da ordem (com os loops invertidos):

```
for i=0,1 do begin
  for j=0,2 do begin
    k=i+j*2
    print,i,j,k,a[i,j]
    do_some_stuff,a[i,j]
  endfor
endfor
```

i	j	**k**	a[i,j]
0	0	**0**	1
0	1	**2**	3
0	2	**4**	5
1	0	**1**	2
1	1	**3**	4
1	2	**5**	6

Há muitas idas e voltas (mostrados pela variável **k**, que indica que lugar na memória foi usado):

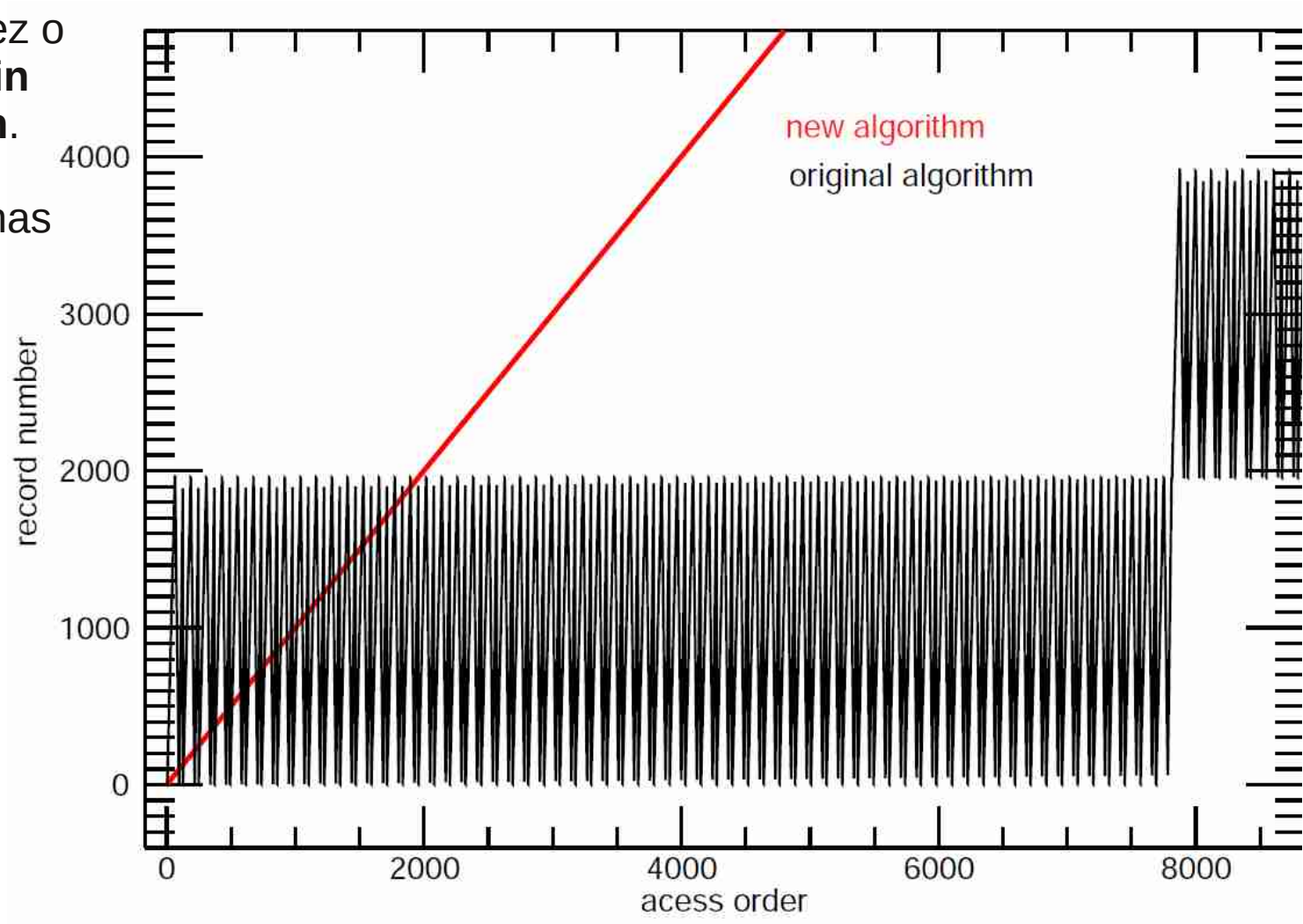


Arrays - armazenamento MD - exemplos

A diferença na ordem de acesso pode ser muito relevante para eficiência.

Neste **exemplo real**, passar a acessar pela ordem de armazenamento (no caso, em disco, onde é mais crítico) fez o programa que tomava **60 min** passar a levar apenas **3 min**.

A mudança afetava ~10 linhas no código-fonte original.



Vetorização - motivação

Computação científica costuma ser fortemente dependente de processar muitos elementos de contêiners – arrays em particular.

Em tempos arcaicos, era necessário o programador escrever explicitamente as operações em termos de cada elemento. Ex. (IDL):

```
for k=0,n1-1 do begin
  for  j=0,n2-1 do begin
    for i=0,n3-1 do begin
      c[i,j,k]=a[i,j,k]+b[i,j,k]
      d[i,j,k]=sin(c[i,j,k])
    endfor
  endfor
endfor
```

O que é ruim por muitos motivos:

- É muito trabalho para escrever o que conceitualmente é apenas **c=a+b, d=sin(c)**.
- Há uma grande possibilidade de erros: erros de digitação, erros no uso dos índices: É **c[i,j,k]** ou **c[k,j,i]**, ou **c[j,k,i]**? Quais são os limites das dimensões? Quais são os índices das dimensões?
- É uma execução serial (um elemento por vez) de uma operação que poderia ser (automaticamente) paralelizada.
- **Em algumas linguagens (dinâmicas, em particular) é extremamente ineficiente**.

Este ainda é um exemplo extremamente simples; em operações mais complicadas (adiante), a possibilidade de erro e a verbosidade desnecessária são muito mais relevantes.

Vetorização - motivação

Qual é a alternativa a escrever todos estes loops em contêiners?

Vetorização - expressões são escritas como a operação pretendida entre as entidades (os contêiners), da mesma forma que se faz em linguagem verbal ou matemática:

c=**a**+**b**
d=sin(**c**)

Onde **a** e **b** são arrays de qualquer dimensão (**a** e **b** têm as mesmas dimensões)

$\mathbf{x} = A \cdot y = U^T U = U \Sigma V^T \cdot \mathbf{y}$

$\mathbf{F} = q\,(\mathbf{E} + \mathbf{v} \times \mathbf{B})$

É o trabalho do compilador / interpretador realizar as operações sobre os elementos, mantendo a contabilidade dos índices.

Em geral, qualquer tarefa onde seja necessário manter a contabilidade de muitas coisas (como índices e dimensões) é adequada para computadores, não para pessoas.

Porque computadores têm memória perfeita (nunca se esquecem ou confundem) e muito grande, e podem iterar muito rápido sobre muitas coisas.

Vetorização - motivação

O programador só informa operações de mais alto nível - **operações vetoriais**. Ex. (IDL):

```
IDL> a=dindgen(4,3,2)
IDL> b=a+randomu(seed,[4,3,2])*10d0
IDL> help,a,b
A               DOUBLE    = Array[4, 3, 2]
B               DOUBLE    = Array[4, 3, 2]
IDL> c=a+b
IDL> d=sin(c)
IDL> help,c,d
C               DOUBLE    = Array[4, 3, 2]
D               DOUBLE    = Array[4, 3, 2]
IDL> A=dindgen(3,3)
IDL> y=dindgen(3)
IDL> x=A#y ;Matrix product of matrix A (3,3) and vector y (3)
IDL> help,y,A,x
Y               DOUBLE    = Array[3]
A               DOUBLE    = Array[3, 3]
X               DOUBLE    = Array[3]
```

Quando compiladores / interpretadores encontram operações assim, eles sabem qual é a idéia pretendida: sabem sobre que elementos têm que operar de que forma.

O software pode automaticamente paralelizar a execução.

É o mesmo que pessoas fariam, na mão: já se sabe que se trata de fazer o mesmo para cada elemento, não é necessário depois de cada elemento decidir o que fazer; se há várias pessoas, cada uma faz um pedaço.

Vetorização - possibilidades

O nível de vetorização suportado varia drasticamente entre linguagens. Da mais ingênua para a mais capaz:

- C, Fortran < 90 e Perl nada têm.
- Fortran 90, 95, 2003 e 2008, C++, Java: vetorização simples:
  - Em C++ e Java, razoável para 1D, desajeitada para mais de 1D (pior ainda para mais de 2D), limitada a operações sobre arrays inteiros ou fatias regulares (mostrada adiante). Em Fortran, melhor suporte para MD, melhorando do 90 ao 2008.
- Biblioteca **Boost** (não padrão) para C++ provê boa funcionalidade para linguagens estáticas. Partes dela devem ser gradualmente incluídas nos próximos padrões.
- R tem melhor suporte a operações não triviais, especialmente até 2D (classe matrix).
- **IDL e Python+Numpy*** têm as operações de mais alto nível (especialmente para mais de 1D, inclusive com números arbitrários de dimensões), que dão muito mais poder e conveniência, eliminando muitos loops.

Algumas funcionalidades mostradas adiante só existem no nível de IDL e Python+Numpy.

*Numpy não é (ainda) parte das bibliotecas padrão de Python. Numpy possivelmente é a biblioteca vetorial mais avançada atualmente, mas Python sem Numpy está em um nível entre o do C++ / Java e o nível do R.

Vetorização - paralelização

Operações vetoriais contém toda a informação necessária para paralelizar a tarefa.

Compiladores e interpretadores podem usar esta informação para gerar código paralelo:

- Para um núcleo: uso de instruções vetoriais do processador:
  - MMX - Pentium
  - SSE - Streaming SIMD Extensions - versões 1 a 4.2 (Netburst (Pentium III) a Nehalem (Core i7 e Xeon), parcialmente em AMD) e SSE4a Barcelona (AMD Opteron)
  - AVX (Advanced Vector Extensions) – Sandy Bridge, Ivy Bridge (Pentium a Core i7 e Xeon), Bulldozer (AMD Opteron).

  Em geral habilitado por opções `-O2`, `-O3`, `-vec`, `-vect`, `-fast`, `-fastsse`, `-Ofast` em compiladores.

- Para vários núcleos: uso de vários threads
  - **`workshare`** em OpenMP (próxima aula)
  - Thread pool (IDL)
  - Em geral habilitado por opções `-parallel`, `-ftree-parallelize-loops` em compiladores.

Vetorização avançada (mas essencial) - Índices 1D x MD

Quando um array tem mais de 1D (MD), elementos podem ser selecionados pelos M índices. Ex. (IDL):

```
IDL> a=bindgen(4,3)*2
IDL> print,a
       0       2       4       6
       8      10      12      14
      16      18      20      22
IDL> print,a[1,2]
      18
```

Ou por apenas um índice, que é a posição do elemento na ordem interna de armazenamento*:

```
IDL> print,a[9]
      18
IDL> print,array_indices(a,9)
           1           2
```

O que é de utilidade comum, principalmente em funções que buscam elementos (adiante), que retornam índices 1D. A conversão MD->1D é mais simples:

```
IDL> adims=size(a,/dimensions)
IDL> print,adims
           4           3
IDL> print,a[1+adims[0]*2]
      18
```

Costuma haver funções prontas para fazer estas conversões.

*Essencial saber se o array é *row major* ou *column major*.

Vetorização avançada (mas essencial) - *fancy indexing*

Seleções de elementos por expressões vetoriais (***fancy indexing***, em Numpy):

- Quando se usa apenas um array de índices para outro array, o resultado tem a mesma dimensão que o array de índices (índices 1D), com os elementos correspondentes a cada posição no array de índices:

```
IDL> print,a[[0,1,3,5]]
       0       2       6      10
IDL> print,[[0,1],[3,5]]
       0       1
       3       5
IDL> print,a[[[0,1],[3,5]]]
       0       2
       6      10
```

→ Array 1D, 4 elementos: 0,1,3,5 de a

→ Array 2D de índices (1D), 4 elementos

→ Array 2D, 4 elementos de a, dados pelos índices (1D) acima.

Da mesma forma poderia se usar como índices um array 3D, 4D, etc, que geraria um resultado das mesmas dimensões do array de índices: não importa a dimensão de `a`.

•Quando cada dimensão recebe um array de índices, os arrays de índices têm que ter dimensões iguais. O resultado tem estas dimensões, com os elementos selecionados pelo índice de cada dimensão na posição correspondente:

```
IDL> print,a[[0,1],[3,5]]
      16      18
```

→ Array 1D, 2 elementos:
0: `a[0,3]`
1: `a[1,5]`

Vetorização avançada (mas essencial)

Operandos de dimensões diferentes: Operações vetoriais não se limitam a aplicar operações elemento-a-elemento entre arrays de mesma forma:

Operações com formas diferentes ocorrem por regras de conformidade: operações são permitidas entre arrays de dimensões **conformes** (em Numpy, dimensões que podem ser *broadcast* para se tornar iguais):

- Escalares sempre são confomes a qualquer array: o escalar é aplicado elemento a elemento.

- Se os arrays não têm dimensões iguais, o tratamento varia:
  - **IDL**: é usado o menor comprimento de cada dimensão, ignorando (truncado) o que sobra nos arrays onde as dimensões são maiores:

```
IDL> b=[0,1,2]

IDL> c=[1,2,3,4]

IDL> print,b*c
       0       2       6

IDL> print,c*2
       2       4       6       8
```

Vetorização avançada (mas essencial)

Operandos de dimensões diferentes: Operações vetoriais não se limitam a aplicar operações elemento-a-elemento entre arrays de mesma forma:

Numpy (Python): se em uma dimensão um array tem dimensão 1, este array tem seus elementos repetidos naquela dimensão até se tornar da mesma dimensão que o outro array.

```
>>> a = array([[1,2,3],[4,5,6]])
>>> b=array([[1],[2]])
>>> print(a)
[[1 2 3]
 [4 5 6]]
>>> print(b)
[[1]
 [2]]
>>> print(a+b)
[[2 3 4]
 [6 7 8]]
```

Ou seja, em IDL o resultado tem em cada dimensão o tamanho mínimo entre os operandos; em Numpy, o tamanho é o máximo entre os operandos (os de dimensão menor são repetidos até as dimensões ficarem iguais).

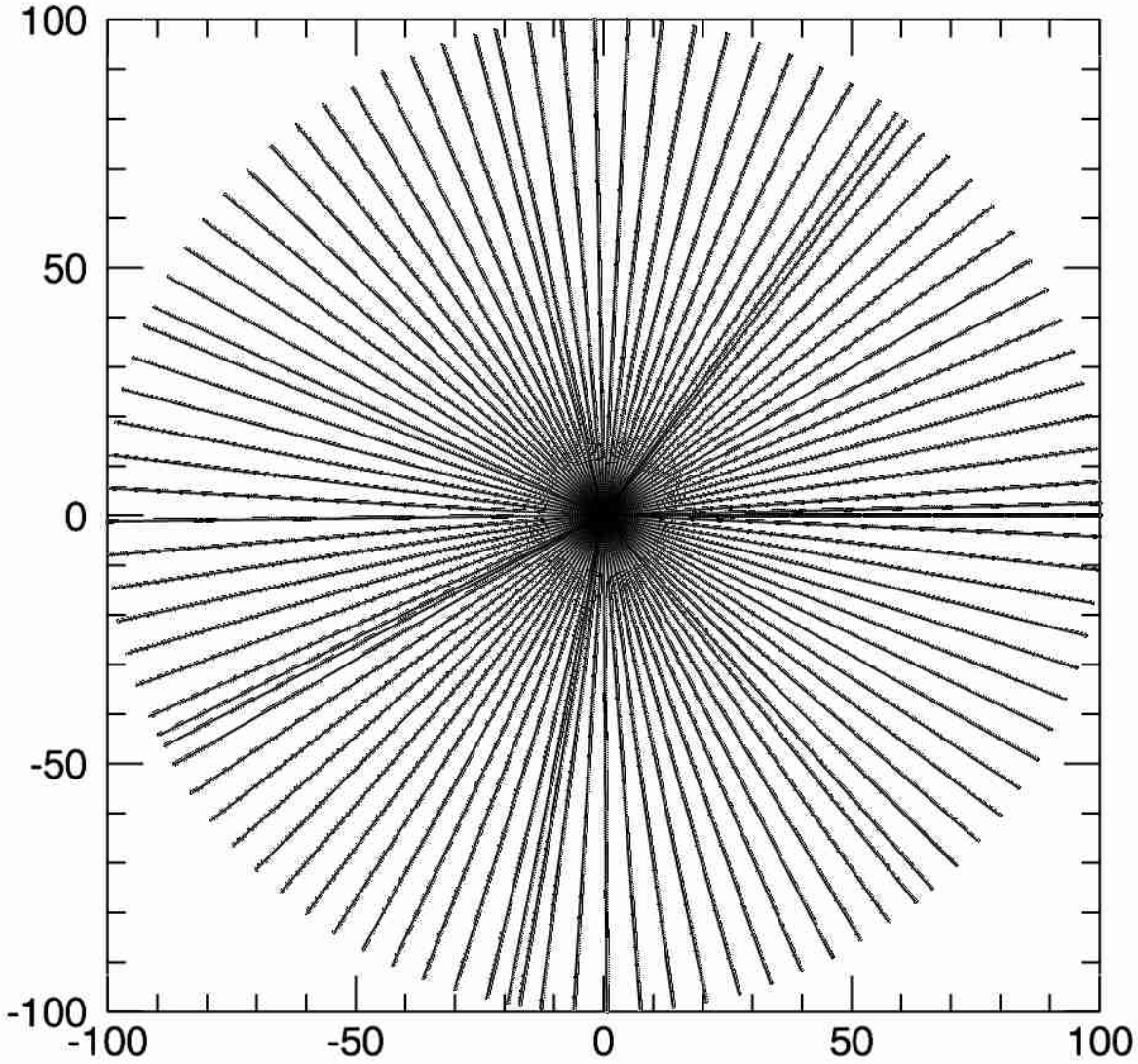
R, os elementos do array menor são repetidos o suficiente, ciclicamente:

```
> x=c(1,2,3)+c(10,20)
Warning message:
In c(1, 2, 3) + c(10, 20) :
  longer object length is not a multiple of shorter object length
> x
[1] 11 22 13
```

Vetorização avançada (mas essencial) - redimensionamento

É comum ter que mudar as dimensões de um array:

•Para conformidade com outros arrays. Ex: converter as coordenadas em uma grade polar (**r, theta**) para cartesianas, para obter os valores de uma função (**temperature**) em coordenadas cartesianas:



Vetorização avançada (mas essencial) - redimensionamento

Ex. (IDL):

```
IDL> help,temperature,r,theta,nr,ntheta
TEMPERATURE     FLOAT     = Array[200, 100]
R               DOUBLE    = Array[200]
THETA           DOUBLE    = Array[100]
NR              LONG      =          200
NTHETA          LONG      =          100

IDL> r_2d=rebin(r,nr,ntheta)
IDL> theta_2d=rebin(reform(theta,1,ntheta),nr,ntheta)
IDL> print,r_2d[0:3,0:2]
      0.10000000      0.60251256       1.1050251       1.6075377
      0.10000000      0.60251256       1.1050251       1.6075377
      0.10000000      0.60251256       1.1050251       1.6075377

IDL> print,theta_2d[0:3,0:2]
       0.0000000       0.0000000       0.0000000       0.0000000
       3.6000000       3.6000000       3.6000000       3.6000000
       7.2000000       7.2000000       7.2000000       7.2000000

IDL> x=r_2d*cos(theta_2d*!dpi/18d1)
IDL> y=r_2d*sin(theta_2d*!dpi/18d1)
IDL> help,r_2d,theta_2d,x,y
R_2D            DOUBLE    = Array[200, 100]
THETA_2D        DOUBLE    = Array[200, 100]
X               DOUBLE    = Array[200, 100]
Y               DOUBLE    = Array[200, 100]
```

Cada coluna corresponde a um raio, e cada linha a um ângulo, da mesma forma que **`temperature`**; a mesma relação será gerada para **`x`** e **`y`**, calculados para cada ponto do array **`temperature`**.

Vetorização avançada (mas essencial) - redimensionamento

•Para reduções. Ex: dadas n (4) coordenadas de pontos em 2D (cada linha tem x e y de cada ponto). Ex. (IDL):

```
IDL> xy=[[3d0,4d0],[6d0,8d0],[9d0,12d0],[12d0,16d0]]
IDL> print,xy
       3.0000000       4.0000000
       6.0000000       8.0000000
       9.0000000       12.000000
       12.000000       16.000000
```

Cada linha tem as coordenadas de um ponto

De onde se quer calcular a distância à origem de cada ponto:

Dimensão sobre a qual é realizada a soma.

```
IDL> r=sqrt(total(xy^2,1))
IDL> print,r
       5.0000000       10.000000       15.000000       20.000000
```

O mesmo algorítimo funcionaria, idêntico, se os pontos fossem em 3D:

```
IDL> print,xyz
       0.0000000       1.0000000       2.0000000
       3.0000000       4.0000000       5.0000000
       6.0000000       7.0000000       8.0000000
       9.0000000       10.000000       11.000000
IDL> r=sqrt(total(xyz^2,1))
IDL> print,r
       2.2360680       7.0710678       12.206556       17.378147
IDL> print,r^2
       5.0000000       50.000000       149.00000       302.00000
```

Vetorização avançada (mas essencial) - redimensionamento

Outras transformações (comuns para armazenamento / processamento):

Ex. (IDL): Uma rotina vetorial que processa n pontos em 3D, como arrays `[3,n]`. Mas os n pontos são coordenadas calculadas para cada posição em uma imagem, colocados em um array 3D. Ex. (IDL):

```
IDL> help,xyz
XYZ             FLOAT     = Array[3, 1000, 2000]
```

É necessário converter **`xyz`** de **`[3,ns,nl]`** para **`[3,ns*nl]`**, para poder usar na rotina:

```
IDL> sz=size(xyz,/dimensions)
IDL> print,sz
           3        1000        2000
IDL> xyz_2d=reform(xyz,sz[0],sz[1]*sz[2])
IDL> help,xyz_2d
XYZ_2D          FLOAT     = Array[3, 2000000]
IDL> r_theta_phi_2d=convert_to_spherical(xyz_2d)
```

O resultado tem a mesma forma de xyz_2d:

```
IDL> help,r_theta_phi_2d
R_THETA_PHI_2D  FLOAT     = Array[3, 2000000]
```

É necessário converter de volta a 3D:

```
IDL> r_theta_phi=reform(r_theta_phi_2d,sz[0],sz[1],sz[2])
IDL> help,r_theta_phi
R_THETA_PHI     FLOAT     = Array[3, 1000, 2000]
```

Vetorização avançada (mas essencial) - buscas

Buscas: **encontrar em um array elementos pelas suas propriedades** é uma das operações mais comuns. O que é muito facilitado por semântica vetorial. Ex. (IDL):

•Filtros*:

```
w=where((spectrum.wavelength gt 4d3) and (spectrum.wavelength lt 6d3),/null)
spectrum=spectrum[w]
```

(seleciona apenas os elementos de **`spectrum`** onde o campo **`wavelength`** tem valores entre 4d3 e 6d3)

```
spectrum=spectrum[where(finite(spectrum.flux),/null)]
```

(seleciona apenas os elementos de **`spectrum`** onde o campo **`flux`** não é **NaN** ou **infinito**)

•Elementos específicos*:

```
w=where(observations.objects eq 'HD3728',/null)
p=plot(observations[w].wavelength,observations[w].flux)
```

Se este tipo de operação tem que ser feito com vários ou todos os nomes de objetos, pode ser mais apropriado armazenar os objetos em um hash (onde o acesso é por nome), do que em um array, como acima:

```
p=plot((observations['HD3278']).wavelength,(observations['HD3278']).flux)
```

*Semelhante ao uso de **`WHERE`** em SQL.

Vetorização avançada (mas essencial) - buscas

Buscas: **encontrar em um array elementos pelas suas propriedades** é uma das operações mais comuns. O que é muito facilitado por semântica vetorial. Ex. (IDL):

•Elementos mais próximos de um valor:

- Muito necessário para encontrar elementos com valores reais (não inteiros), já que pode não haver elementos de valores exatamente iguais. Ex: encontrar qual elemento do array se trata da linha Hα:

```
halpha=6562.8d0
!null=min(lines.wavelength-halpha,minloc,/absolute)
do_some_stuff,lines[minloc]
```

Índice do elemento onde ocorre o mínimo

O mínimo procurado é dos módulos

•Local em uma seqüência monotônica que contém um valor.

- Ex: Em um modelo, alterar a temperatura nas células da grade que contém um certo raio (**r_search**):

```
IDL> help,temperature,r,theta,phi,r_search
TEMPERATURE     DOUBLE    = Array[300, 100, 200]
R               DOUBLE    = Array[300]
THETA           DOUBLE    = Array[100]
PHI             DOUBLE    = Array[200]
R_SEARCH        DOUBLE    =       74.279000
IDL> print,minmax(r)
      17.485000       100.00000
IDL> w=value_locate(r,r_search)
IDL> print,w,r[w],r[w+1]
         205       74.058829       74.334799
IDL> temperature[w,*,*]=some_other_temperature
```

Retorna o índice onde `r` (array ordenado) envolve o valor `r_search` (no caso, escalar, mas se fossem procurados vários, poderia ser um array).

Vetorização avançada (mas essencial) - inversão de índices

Com freqüência é necessário determinar quais são os índices onde cada valor ocorre em um array.

Necessário para ordenar / classificar / selecionar vários elementos de um array pelos seus valores.:

- Dados vários arquivos de observações:
  - Identificar quais foram feitas em cada data onde houve observações - ordem por inteiros (datas julianas) não necessariamente contíguos (possivelmente com intervalos de muitas unidades), não iniciados em 0.

  - ➔ Identificar quais observações são de cada objeto / instrumento - ordem por strings.

- Dados vários modelos, identificar quais têm um dos parâmetros em cada caixa (faixa de valores).

É o mesmo que realizar uma busca para cada valor diferente de um array.

Mas fazer uma busca para cada valor é ineficiente, pois se está varrendo o array N vezes (havendo N valores diferentes).

Uma busca combinada só varre o array uma vez, e vai mantendo um registro do que encontra. É muito mais eficiente, e mais fácil de fazer.

Vetorização avançada (mas essencial) - inversão de índices

Semelhante ao uso de **GROUP BY** em SQL:

id	date	target
1	2011/01/07	HD36052
2	2011/01/07	HD28655
3	2011/01/08	HD18694
4	2011/01/10	HD36052
5	2011/01/10	HD36052
6	2011/01/10	HD18694

→ Tabela **obs**

```
SELECT target, COUNT(target) AS count
  FROM obs
  GROUP BY target
  ORDER BY count
```

target	COUNT(target)
HD28655	1
HD18694	2
HD36052	3

→ Resultado

Mas mantendo a informação de quais elementos entram em cada grupo (não só quantos).

Vetorização avançada (mas essencial) - inversão de índices

Ex. (IDL): encontrar quais observações foram feitas em cada data do conjunto:

```
IDL> help,obs
OBS             STRUCT    = -> <Anonymous> Array[10]

IDL> print,obs[0]
{ HD28985     2455563 sl_0092.fits<ObjHeapVar4(HASH)>}

IDL> help,obs[0]
** Structure <e41dc4b8>, 4 tags, length=48, data length=40, refs=3:
   OBJECT          STRING    'HD28985'
   DATE            LONG           2455563
   FILE            STRING    'sl_0092.fits'
   DATA            OBJREF    <ObjHeapVar4(HASH)>

IDL> print,obs.object
HD28985 HD59382 HD63281 HD63281 HD48561 HD78325 HR892561 HR748267
HR189365 HR167382

IDL> print,obs.date
     2455563     2455569     2455569     2455570     2455570
2455570     2455570     2455570     2455574     2455576
```

Vetorização avançada (mas essencial) - inversão de índices

Inversão é melhor realizada por algorítimos de histograma: a cada elemento do array, se verifica a que bin ele pertence, e se anota naquele bin o índice daquele elemento.

```
IDL>
h=histogram_pp(obs.date,min=min(obs.date),binsize=1L,reverse_list=rl,reverse_h
ash=rh,locations=locations)
```

Cada bin tem o número de observações de cada data:

```
IDL> foreach element,locations,i do print,element,' : ',h[i]
     2455563 :            1
     2455564 :            0
     2455565 :            0
     2455566 :            0
     2455567 :            0
     2455568 :            0
     2455569 :            2
     2455570 :            5
     2455571 :            0
     2455572 :            0
     2455573 :            0
     2455574 :            1
     2455575 :            0
     2455576 :            1
```

*__histogram_pp()__ é uma interface para a função __histogram()__ da biblioteca padrão, para fornecer os índices reversos em listas e/ou hashes, no lugar dos complicados arrays de __histogram()__. Pode ser encontrada em http://ppenteado.net/idl/pp_lib/doc/index.html

Vetorização avançada (mas essencial) - inversão de índices

Mas pouco interessa saber **quantas** são as observações de cada data. O objetivo é saber **quais** são de cada data. Para isso servem os *índices reversos* gerados na criação do histograma. Em forma de lista:

```
IDL> help,rl
RL              LIST  <ID=44  NELEMENTS=14>
IDL> foreach element,locations,i do print,element,' : ',rl[i]
   0      2455563 :            0
   1      2455564 : !NULL
   2      2455565 : !NULL
   3      2455566 : !NULL
   4      2455567 : !NULL
   5      2455568 : !NULL
   6      2455569 :            1           2
   7      2455570 :            3           4           5           6           7
   8      2455571 : !NULL
   9      2455572 : !NULL
  10      2455573 : !NULL
  11      2455574 :            8
  12      2455575 : !NULL
  13      2455576 :            9
```

Cada elemento da lista **`rl`** é um (possivelmente vazio) array, que tem os índices dos elementos (no caso, do array **`obs.date`**) que caem no bin correspondente. Ex: na data **2455569** houve duas observações, as de índices **1** e **2** no array **`obs`**:

```
IDL> print,locations[6],obs[rl[6]]
     2455569
{ HD59382     2455569 sl_0102.fits<ObjHeapVar8(HASH)>}
{ HD63281     2455569 sl_0110.fits<ObjHeapVar12(HASH)>}
```

Vetorização avançada (mas essencial) - inversão de índices

Para alguns usos, pode ser mais conveniente usar os índices em forma de hash:

```
IDL> help,rh
RH              HASH  <ID=83  NELEMENTS=14>
IDL> print,rh
2455566: !NULL
2455564: !NULL
2455572: !NULL
2455575: !NULL
2455565: !NULL
2455574:            8
2455563:            0
2455567: !NULL
2455570:            3           4           5           6           7
2455576:            9
2455569:            1           2
2455573: !NULL
2455568: !NULL
2455571: !NULL
```

Cada elemento do hash **`rh`** é um (possivelmente vazio) array, que tem os índices dos elementos (no caso, do array **`obs.date`**) que caem no bin correspondente. Ex: na data **2455569** houve duas observações, as de índices **1** e **2** no array **`obs`**:

```
IDL> print,rh[2455569]
           1           2
IDL> print,obs[rh[2455569]]
{ HD59382     2455569 sl_0102.fits<ObjHeapVar8(HASH)>}
{ HD63281     2455569 sl_0110.fits<ObjHeapVar12(HASH)>}
```

Vetorização avançada (mas essencial) - inversão de índices

O mesmo tipo de inversão pode ser feito por arrays que não sejam inteiros:

No caso de faixas uniformes de valores reais, pode-usar diretamente um histograma, escolhendo-se os tamanhos dos bins.

No caso de reais únicos (buscando-se os valores *exatamente* iguais), faixas de valores não uniformes, strings, ou valores muito esparsos (ex: `[1,1000,3007,3008,30010]`), é necessário e/ou conveniente se trabalhar com índices inteiros para valores únicos.

No exemplo anterior, para evitar todos os bins vazios de várias datas seguidas sem observações:

```
IDL> obs=obs[sort(obs.date)]
IDL> u=uniq(obs.date)
IDL> print,u
           0           2           7           8           9
```

Normalmente, funções que identificam valores únicos esperam arrays ordenados

Índices dos elementos únicos

Datas únicas:

```
IDL> udates=obs[u].date
IDL> print,udates
     2455563     2455569     2455570     2455574     2455576
```

Usa-se então, no lugar do array original (`obs.date`), um array de índices para o original:

```
IDL> indexes=value_locate(udates,obs.date)
IDL> print,indexes
           0           1           1           2           2           2
    2           2           3           4
```

Vetorização avançada (mas essencial) - inversão de índices

Há um nível de indireção a mais, mas não há mais bins vazios (e é o que torna possível bins de tamanhos desiguais, bins reais ou de strings):

```
IDL>
h=histogram_pp(indexes,min=min(indexes),binsize=1L,reverse_list=rl,rev
erse_hash=rh,locations=locations)
IDL> print,rh
0:            0
1:            1           2
3:            8
2:            3           4           5           6           7
```

Convertendo os índices para índices do array original:

```
IDL> oindexes=list()
IDL> foreach el,rl do oindexes.add,indexes[el]
IDL> olocations=udates[locations]
IDL> foreach el,rl,i do print,olocations[i],' : ',obs[el].object
     2455563 : HD28985
     2455569 : HD59382 HD63281
     2455570 : HD63281 HD48561 HD78325 HR892561 HR748267
     2455574 : HR189365
     2455576 : HR167382
```

Exercícios - vetorização

Escrever uma função para multiplicação de matrizes.

Exemplo artificial: toda linguagem tem uma função (ou operador) para isso em alguma biblioteca (padrão ou não) .

Equivalente a (ex. IDL):

```
function pp_para_on_exe_1,mat1,mat2
sz1=size(mat1,/dimensions)
sz2=size(mat2,/dimensions)
if sz1[0] ne sz2[1] then begin
  message,'Second dimension of second array must match first dimension
of first array'
  return,ret
endif
ret=dblarr(sz2[0],sz1[1])
for i=0L,sz1[1]-1 do begin
 for j=0L,sz2[0]-1 do begin
   for k=0L,sz1[0]-1 do begin
     ret[j,i]+=mat1[k,i]*mat2[j,k]
   endfor
 endfor
endfor
return,ret
end
```

Uma solução em http://www.ppenteado.net/ast/pp_para_on/pp_para_on_sol_2.pdf

Exercícios - vetorização

Escrever uma função para multiplicação de matrizes.

Exemplo artificial: toda linguagem tem uma função (ou operador) para isso em alguma biblioteca (padrão ou não) .

Equivalente a (ex. Fortran):

```
module pp_para_on_exe_1
implicit none
interface pp_para_on_exe_1_f
  module procedure pp_para_on_exe_1_ff
end interface
contains
function pp_para_on_exe_1_ff(mat1,mat2) result(mat12)
implicit none
integer :: i,j,k,sz1(2),sz2(2)
double precision :: mat1(:,:),mat2(:,:)
double precision,allocatable :: mat12(:,:)
sz1=(/size(mat1,dim=1),size(mat1,dim=2)/)
sz2=(/size(mat2,dim=1),size(mat2,dim=2)/)
if (sz1(2) .ne. sz2(1)) then
  print *,'Second dimension of second array must match first dimension
of first array'
  return
endif
```

Exercícios - vetorização

(continuação)

```
allocate(mat12(sz1(1),sz2(2)))
do i=1,sz2(2)
  do j=1,sz1(1)
    do k=1,sz2(1)
      mat12(j,i)=mat12(j,i)+mat1(j,k)*mat2(k,i)
    enddo
  enddo
enddo
return
end function pp_para_on_exe_1_ff
```

Uma solução em http://www.ppenteado.net/ast/pp_para_on/pp_para_on_sol_2.pdf

Algumas referências

Vetorização

- *Importance of explicit vectorization for CPU and GPU software performance*
  Dickson, Karim e Hamze, Journal of Computational Physics **230**, pp. 5383-5398 (2011)
  http://dx.doi.org/10.1016/j.jcp.2011.03.041

- *Numpy Reference Guide* | *Numpy User Guide*
  http://docs.scipy.org/doc/

- *Python Scripting for Computational Science* (2010)
  Hans Petter Langtangen
  http://www.amazon.com/Python-Scripting-Computational-Science-Engineering/dp/3642093159/

- *Guide to Numpy* (2006)
  Travis E. Oliphant
  http://www.tramy.us/

- *Modern IDL* (2011)
  Michael Galloy
  http://modernidl.idldev.com/

- Artigos sobre vetorização em IDL:
  http://www.idlcoyote.com/tips/rebin_magic.html
  http://www.idlcoyote.com/tips/array_concatenation.html
  http://www.idlcoyote.com/tips/histogram_tutorial.html
  http://www.idlcoyote.com/code_tips/asterisk.html
  http://www.idlcoyote.com/misc_tips/submemory.html

Programa

1 – Conceitos

- Motivação
- Formas de paralelização
  - Paralelismo de dados
  - Paralelismo de tarefas
- Principais arquiteturas paralelas atuais
  - Com recursos compartilhados
  - Independentes
- Paradigmas discutidos neste curso:
  - Vetorização
  - OpenMP
  - MPI
- Escolhas de forma de vetorização
- Algumas referências
- Exercícios – testes de software a usar no curso

Slides em http://www.ppenteado.net/ast/pp_para_on_1.pdf

Exemplos em http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf
pp.penteado@gmail.com

Programa

2 – Vetorização

- Motivação
- Arrays – conceitos
- Organização multidimensional
- Arrays – uso básico
- Arrays – row major x column major
- Operações vetoriais
- Vetorização avançada
  - Operações multidimensionais
  - Redimensionamento
  - Buscas
  - Inversão de índices
- Algumas referências
- Exercícios - vetorização

Slides em http://www.ppenteado.net/ast/pp_para_on_2.pdf

Exemplos em http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf
pp.penteado@gmail.com

Programa

3 – OpenMP

- Motivação
- Características
  - Diretrizes
  - Estruturação
- Construções
  - parallel
  - loop
  - section
  - workshare
- Cláusulas
  - Acesso a dados
  - Controle de execução
- Sincronização
  - Condições de corrida
- Exercícios - OpenMP

Slides em http://www.ppenteado.net/ast/pp_para_on_3.pdf

Exemplos em http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf
pp.penteado@gmail.com

Programa

4 – MPI

- Motivação
- Características
- Estruturação
- Formas de comunicação
- Principais funções
  - Controle
  - Informações
  - Comunicação
- Boost.MPI
- Sincronização
  - Deadlocks
- Exercícios - MPI

Slides em http://www.ppenteado.net/ast/pp_para_on_4.pdf

Exemplos em http://www.ppenteado.net/ast/pp_para_on/
Artigo relacionado: http://www.ppenteado.net/papers/iwcca/iwcca_pfp.pdf

pp.penteado@gmail.com