

Paralelização

Introdução a vetorização, OpenMP e MPI

1 – Conceitos e vetorização

Paulo Penteado
IAG / USP

pp.penteado@gmail.com
http://www.ppenteado.net/ast/pp_para_1.pdf

Programa

1 – Conceitos e vetorização

- 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
- Vetorização
- Algumas referências

Slides em http://www.ppenteadonet/ast/pp_para_1.pdf

2 – OpenMP, MPI, escolhas

- OpenMP
- MPI
- Escolha de forma de paralelização
- Algumas referências

Slides em http://www.ppenteadonet/ast/pp_para_2.pdf

Exemplos em http://www.ppenteadonet/ast/pp_para/

pp.penteadon@gmail.com

Motivação

Como tornar programas mais rápidos?

Motivação

Como tornar programas mais rápidos?

É só esperar ter computadores mais rápidos?

Motivação

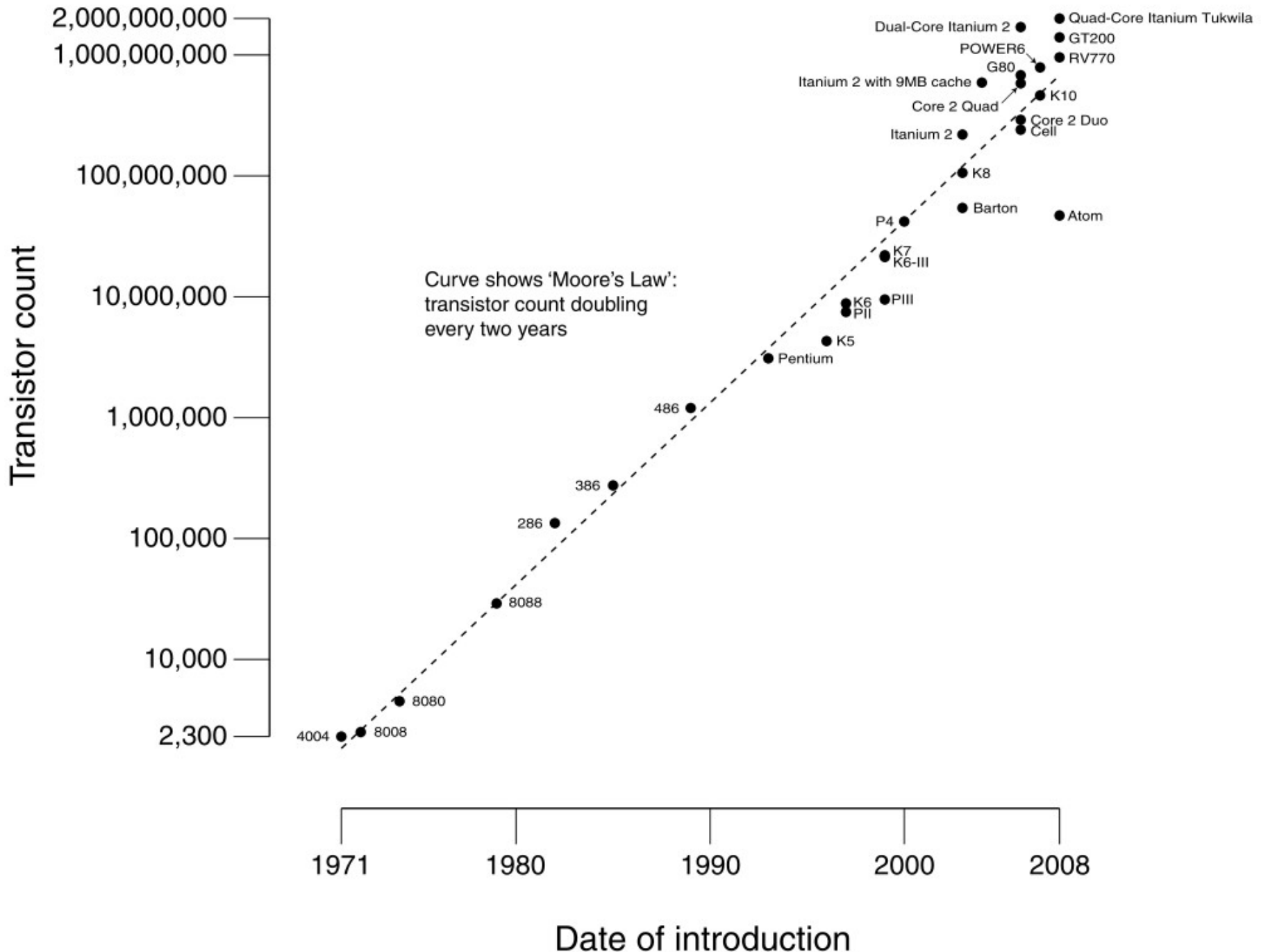
Como tornar programas mais rápidos?

É só esperar ter computadores mais rápidos?

Lei de Moore: *o número de componentes em circuitos integrados tem aumentado exponencialmente, dobrando a cada dois anos.*

- Derivada em 1965, com os dados de 1958 a 1965, prevendo continuar por >10 anos.
- Continua válida, mais de 50 anos depois.

CPU Transistor Counts 1971-2008 & Moore's Law



Motivação

Como tornar programas mais rápidos?

Lei de Moore: *o número de componentes em circuitos integrados tem aumentado exponencialmente, dobrando a cada dois anos.*

- Derivada em 1965, com os dados de 1958 a 1965, prevendo continuar por >10 anos.
- Continua válida, mais de 50 anos depois.

Até 2005 (~ Pentium 4) o aumento de rapidez dos programas era automático: **CPUs mais rápidas tornavam os programas mais rápidos.**

Com o teto em ~3GHz, por motivos térmicos, CPUs tiveram que ser redesenhadas para usar mais núcleos.

- O aumento de capacidade passou a ser predominantemente pelo aumento do número de núcleos.

Além de computadores individuais, muitos (até milhares) de núcleos (CPUs ou GPUs), em computadores interligados ou independentes (clusters, grids, nuvens) formam os supercomputadores.

Motivação

O problema: software que só faz uma coisa de cada vez (***serial, não-paralelizado, sequencial***) não faz uso de múltiplos núcleos.

Portanto, software serial não ganha em tempo com a disponibilidade de vários núcleos.

Não existe um supercomputador feito com um núcleo extremamente rápido.

A única forma de ganhar com o uso de muitos núcleos é fazer mais de uma coisa ao mesmo tempo: **paralelização**.

Paralelização ganhou muita importância recentemente.

Há muitas formas de paralelização, apropriadas a diferentes problemas:

- Algumas delas são mais intuitivas. Outras adicionam muita complexidade para dividir o trabalho entre as unidades e as coordenar.
- Este curso tem apenas algumas das formas mais comuns.

Como fazer mais de uma coisa ao mesmo tempo?

Tradicionalmente, programadores pensam em uma seqüência única (não paralela) de ações.

Identificar como dividir o trabalho em partes que podem ser feitas simultaneamente costuma ser o primeiro obstáculo.

Alguns exemplos da variação de situações:

- Tarefas seriais, a ser executadas para vários conjuntos de dados:
 - Realizar o mesmo processamento em várias observações.
 - Calcular o mesmo modelo para vários parâmetros diferentes.
- Tarefas que contém partes calculadas independentemente:
 - Operações vetoriais (operações sobre arrays, álgebra linear, etc.)
 - Processamento de cada pixel de uma imagem.
 - Cada comprimento de onda em um modelo de transferência radiativa.
 - Cada célula / partícula de um modelo dinâmico.

Como fazer mais de uma coisa ao mesmo tempo?

Tarefas independentes: Ex: um modelo MHD para um objeto astrofísico, após terminar de calcular o estado do sistema em um passo:

(fim do cálculo do estado)

1) Escrever arquivos com o estado do sistema

2) Gerar visualizações do estado do sistema (ex: imagens mostrando temperatura, densidade, etc.)

3) Calcular o espectro observado, para diferentes posições no objeto (para comparar com observações)

4) Calcular as mudanças para o próximo passo:

4a) Forças de contato (pressão, viscosidade, etc.)

4b) Forças à distância:
Gravitacional

Eletromagnética

4c) Forças de radiação

4d) Mudanças nas populações atômicas

(começo do cálculo do próximo estado)

Código serial: uma coisa feita de cada vez

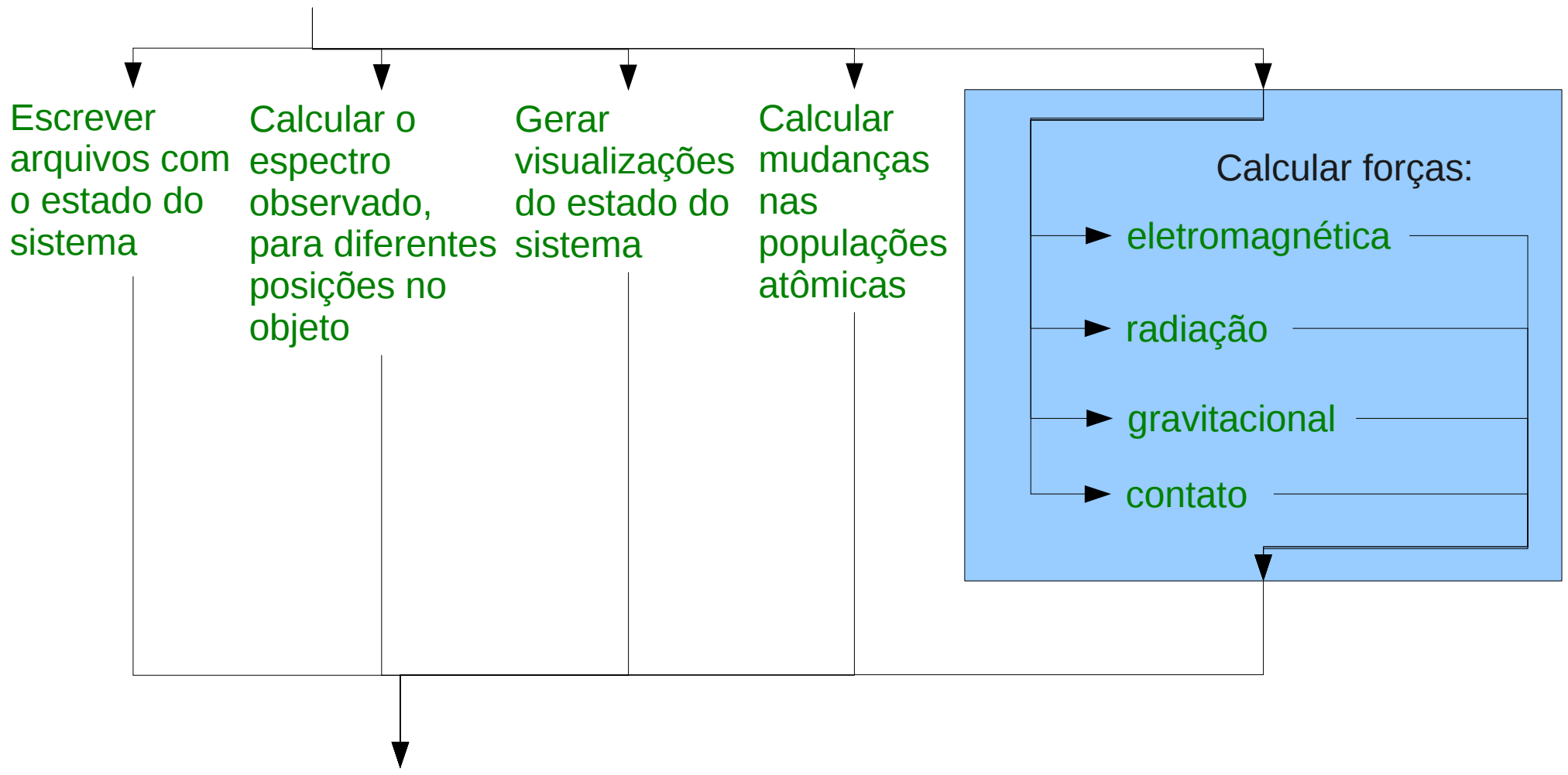
(cada item numerado sendo uma rotina chamada, cada uma após terminar a anterior)

Como fazer mais de uma coisa ao mesmo tempo?

Tarefas independentes: Ex: um modelo MHD para um objeto astrofísico, após terminar de calcular o estado do sistema em um passo:

Código paralelizado: todas as 8 tarefas em verde são feitas ao mesmo tempo

(fim do cálculo do estado)



(começo do cálculo do próximo estado)

Formas de paralelização - classificação

A maior parte dos programadores está habituada em pensar em algoritmos seriais.

Há dificuldade em identificar como paralelizar o programa.

Pode-se pensar em como a mesma tarefa seria realizada manualmente, havendo várias pessoas no lugar de uma só.

Fundamentalmente, a divisão de trabalho pode ser feita no nível de **dados** ou de **tarefas**.

Formas de paralelização - classificação

Paralelismo de dados

Cada unidade processa uma parte dos dados.

Exs:

- Tarefas seriais, a ser executadas para vários conjuntos de dados*:
 - Realizar o mesmo processamento em várias observações.
 - Calcular o mesmo modelo para vários parâmetros diferentes.
- Tarefas que contém partes calculadas independentemente:
 - Operações vetoriais (operações sobre arrays, álgebra linear, etc.)
 - Processamento de cada pixel de uma imagem.
 - Cada comprimento de onda em um modelo de transferência radiativa.
 - Cada célula / partícula de um modelo dinâmico.

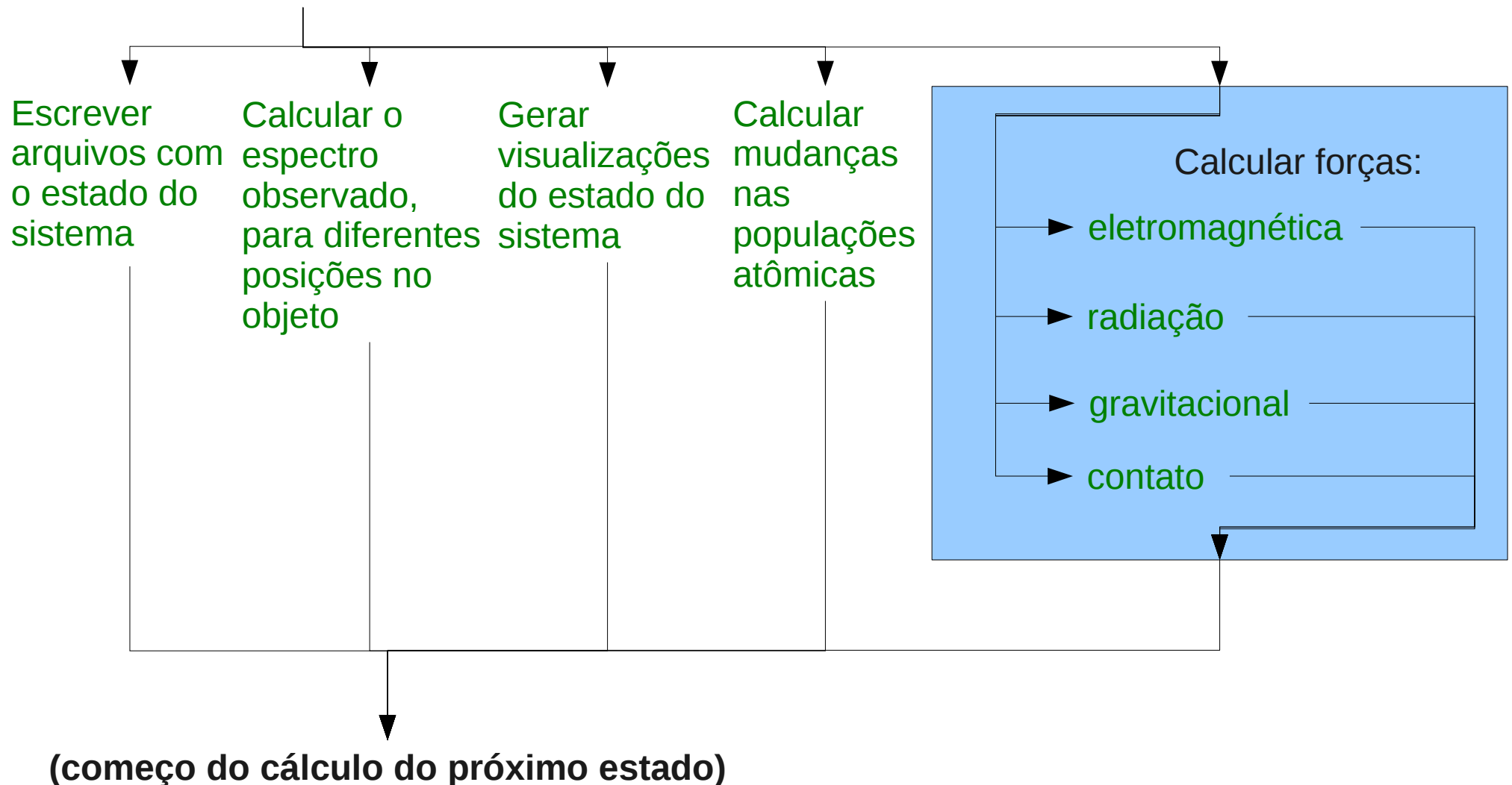
*Este caso pode ser feito por **paralelização extrínseca**: apenas executando ao mesmo tempo várias instâncias de um programa serial, cada uma usando dados diferentes.

Paralelismo de tarefas

Cada unidade realiza uma tarefa independente (sobre dados distintos).

Ex: um modelo MHD para um objeto astrofísico, após terminar de calcular o estado do sistema em um passo:

(fim do cálculo do estado)



Formas de paralelização - classificação

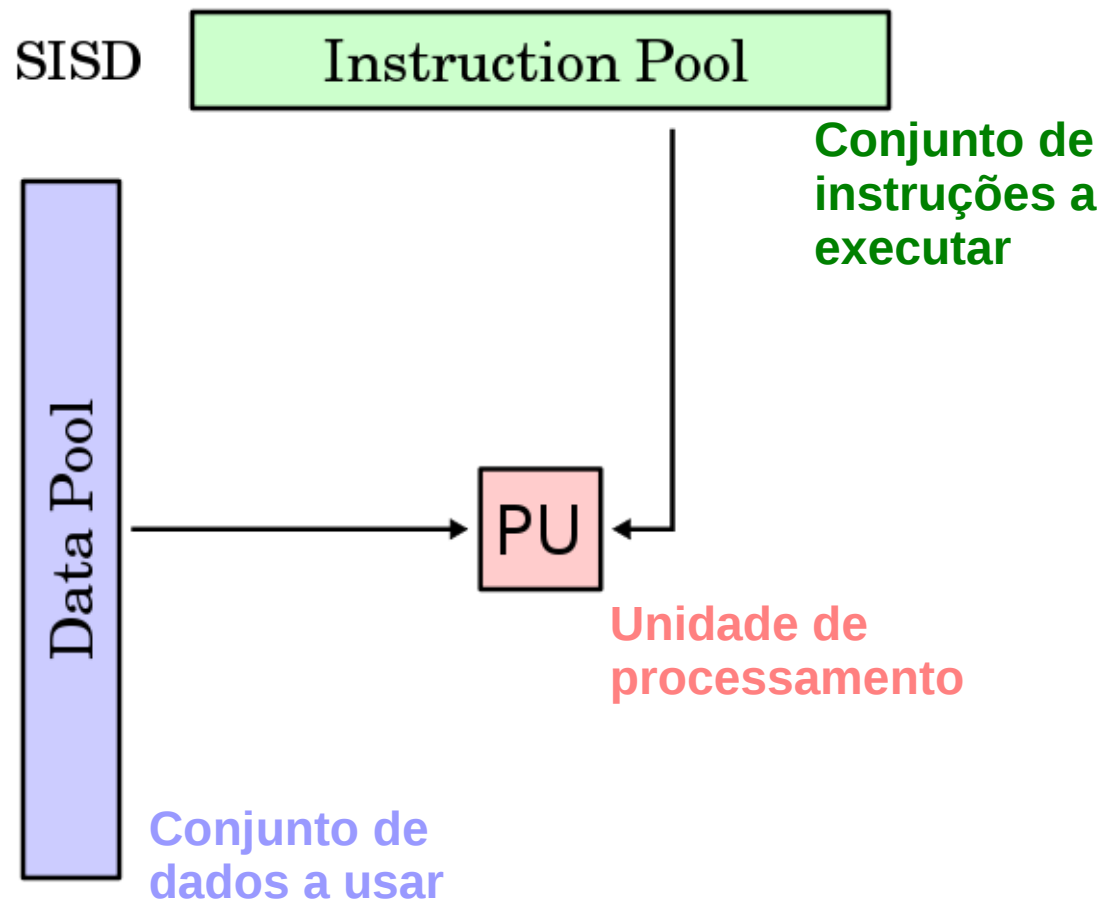
Taxonomia de Flynn (1966):

(Single | Multiple) **Instruction**, (Single | Multiple) **Data**

1) SISD – Single Instruction, Single Data

Programas seriais.

A cada momento, apenas uma instrução é executada, em apenas um elemento dos dados.



Formas de paralelização - classificação

Taxonomia de Flynn (1966):

(Single | Multiple) **Instruction**, (Single | Multiple) **Data**

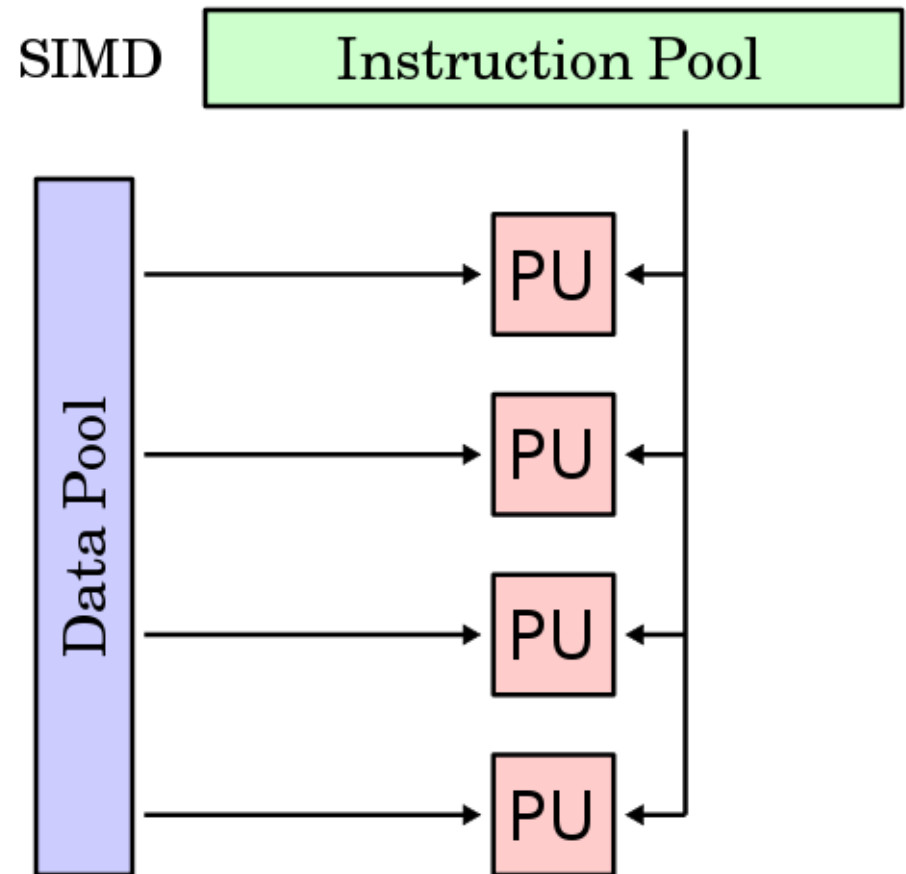
2) SIMD – Single Instruction, Multiple Data

Paralelismo de dados.

A cada momento, apenas uma instrução é executada, em vários elementos do conjunto de dados, simultaneamente.

Exs:

- Operações vetorizadas (adiante)
- Alguns usos de OpenMP e MPI (próxima aula)
- GPUs (em outro curso)



Formas de paralelização - classificação

Taxonomia de Flynn (1966):

(Single | Multiple) **Instruction**, (Single | Multiple) **Data**

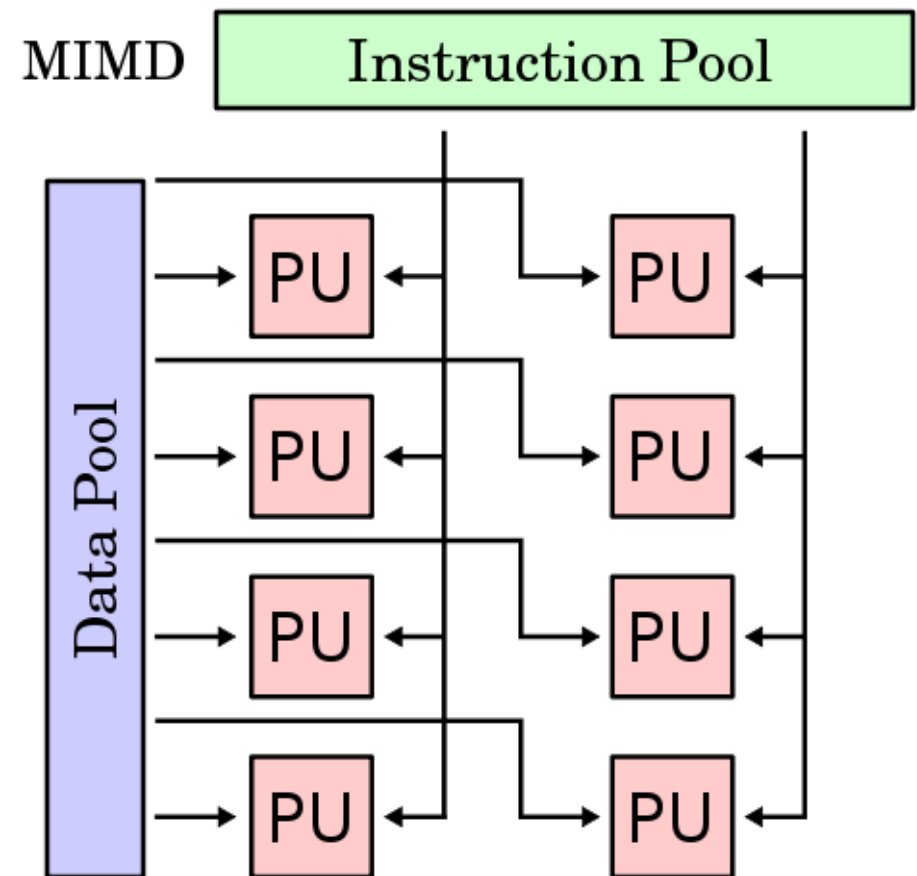
3) MIMD – Multiple Instruction, Multiple Data

Paralelismo de tarefas.

A cada momento, várias instruções são executadas, em vários elementos do conjunto de dados, simultaneamente.

Exs:

- Alguns usos de OpenMP e MPI (próxima aula)
- Paralelização extrínseca (vários programas executados simultaneamente, cada um processando dados diferentes).



Limites à paralelização

Algumas tarefas, intrinsecamente, não são paralelizáveis: cada parte depende das anteriores, não sendo possível as fazer simultaneamente.

Nestes casos, a única possibilidade de paralelização é produzir várias unidades independentes (processar várias imagens, calcular vários modelos) simultaneamente (paralelismo extrínseco).

Exemplo clássico de problema não paralelizável:

- *Não importa quantas pessoas possam ajudar, não vai levar menos que ~9 meses para produzir um bebê.*
- Mas é possível **N** mulheres, simultaneamente, produzirem um bebê cada: ainda leva 9 meses para terminar, mas **N** unidades serão produzidas neste tempo; o tempo médio por unidade é **N** vezes menor.

A maior parte dos problemas em ciências computacionais não tem uma característica única:

- Tipicamente, têm partes não paralelizáveis, e partes de tipos diferentes de paralelização.

É importante **medir** que partes são mais relevantes ao tempo de execução:

- **Sem medir, é fácil se enganar sobre que parte é mais pesada.**
- **A medição deve ser feita com casos representativos:** os gargalos podem mudar entre um caso simples de teste e um caso pesado de uso real.

Limites à paralelização

Paralelizar em N unidades não necessariamente diminui em N vezes o tempo médio para obter cada resultado.

Paralelizar tem um custo (*overhead*), de tarefas adicionais geradas pela paralelização. Exs:

- Dividir o trabalho entre unidades
- Iniciar cada unidade
- Comunicação entre as unidades
- Juntar os resultados ao final
- Finalizar cada unidade

Para que haja um ganho, estes trabalhos adicionais devem ser pequenos em comparação ao trabalho paralelizado.

Em paralelização de tarefas, o número máximo de unidades é determinado pelo algoritmo:

- Não adianta ter 10 mil núcleos, se só há 8 tarefas a fazer simultaneamente.
- Mas é possível (e comum) que cada tarefa possa usar, internamente, paralelização de dados, fazendo uso de várias unidades por tarefa.

Limites à paralelização

Em paralelização de dados a princípio é possível usar qualquer número de unidades.

Mas em casos não completamente paralelizados o ganho não é linear com o número de unidades.

Lei de Amdahl

O modelo mais simples para o ganho de tempo com paralelização.

O ganho de tempo (*speedup*) de um programa com fração **P** paralelizada, executado com **N** unidades:

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

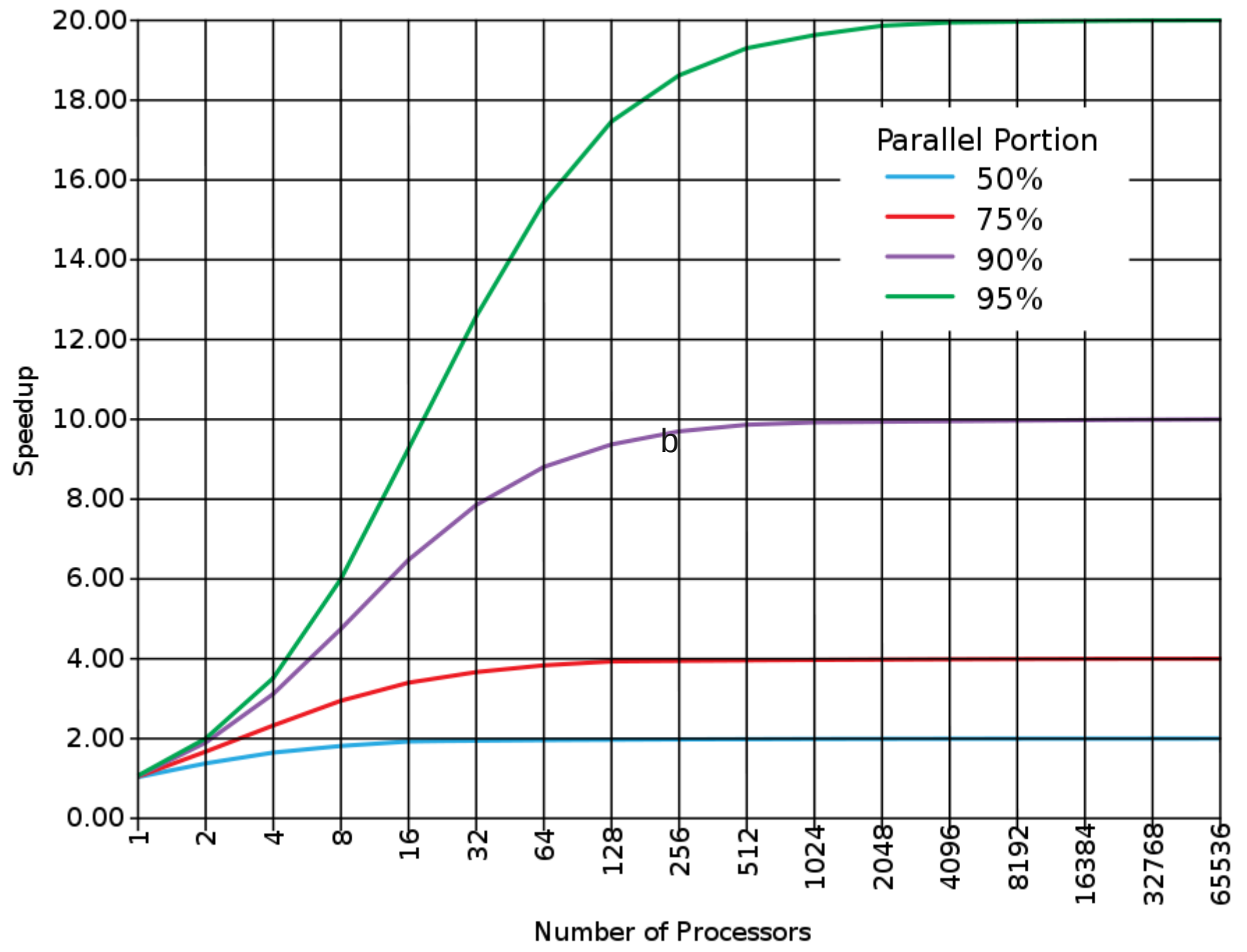
Só considera a relação entre o tempo gasto na parte serial e o tempo gasto na parte paralela.

Não considera custos adicionais (*overheads*) de paralelização.

Útil para prever ganhos, e medir a fração paralelizada de um programa.

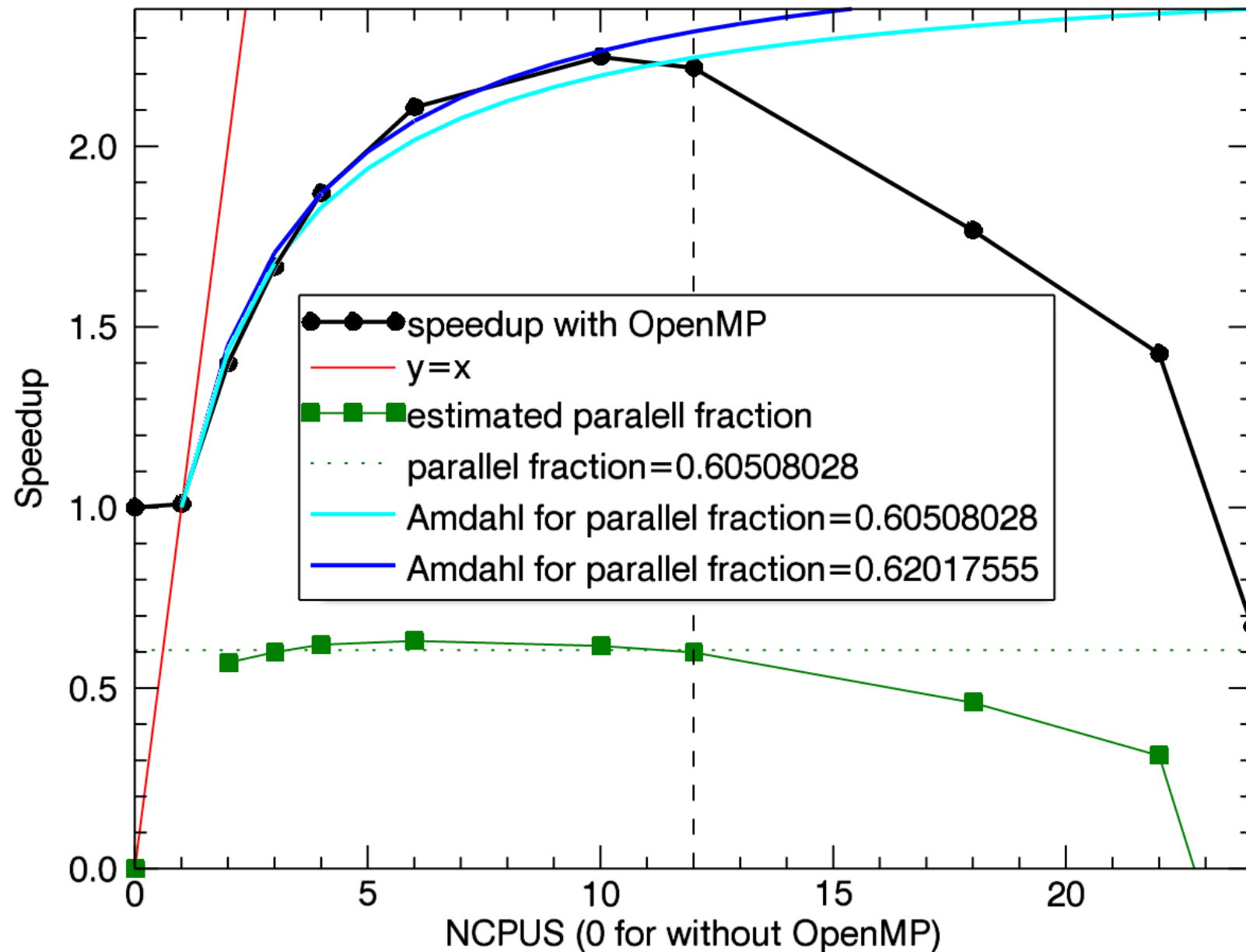
Lei de Amdahl – exemplos teóricos

Amdahl's Law



Lei de Amdahl – exemplo real de medida

time(0)=2.0991667 h, min time=0.93416667 h



O pico em ~10 (11 não foi testado) se deve a o computador ter 12 núcleos: com mais que um processos por unidade de execução, o *overhead* tende a aumentar.

Principais arquiteturas paralelas atuais

A classificação mais fundamental para as arquiteturas de paralelização está no compartilhamento de recursos.

Em alguns casos, sistemas de arquivos (discos, etc.) podem ser compartilhados: todas as unidades têm acesso aos mesmos arquivos.

A memória (RAM) normalmente é o mais importante recurso:

- É muito mais rápida (ordens de magnitude) que discos.
- Normalmente é onde ficam todas as variáveis que um programa usa.
- É muito mais cara (por isso mais escassa) que discos.
- Só raros (e caros) sistemas conseguem compartilhar memória entre nós (computadores) diferentes. Mesmo quando é possível, o acesso à memória entre nós é mais lento que dentro de um nó (por limitações físicas).

Quando há compartilhamento, **o mesmo recurso está disponível a todas as unidades de execução**. Todas podem (a princípio) acessar os mesmos dados, ao mesmo tempo:

Principais arquiteturas paralelas atuais

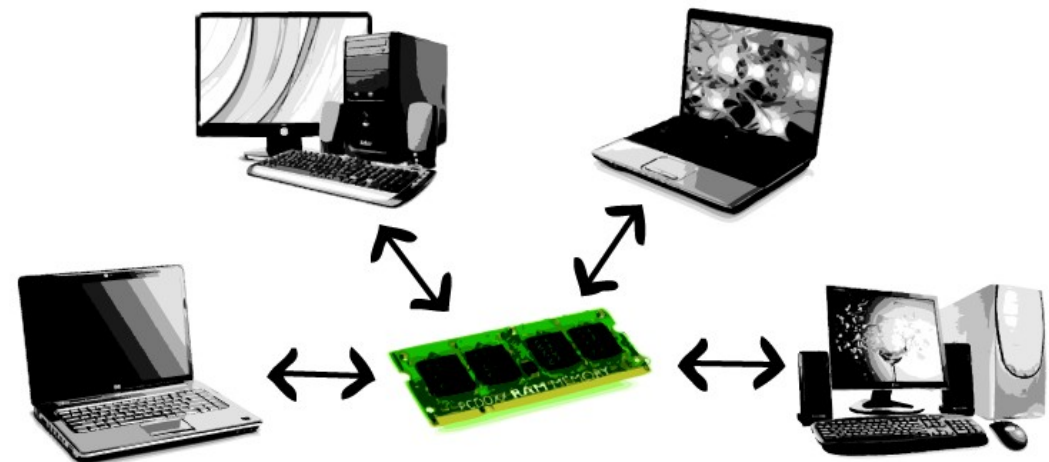
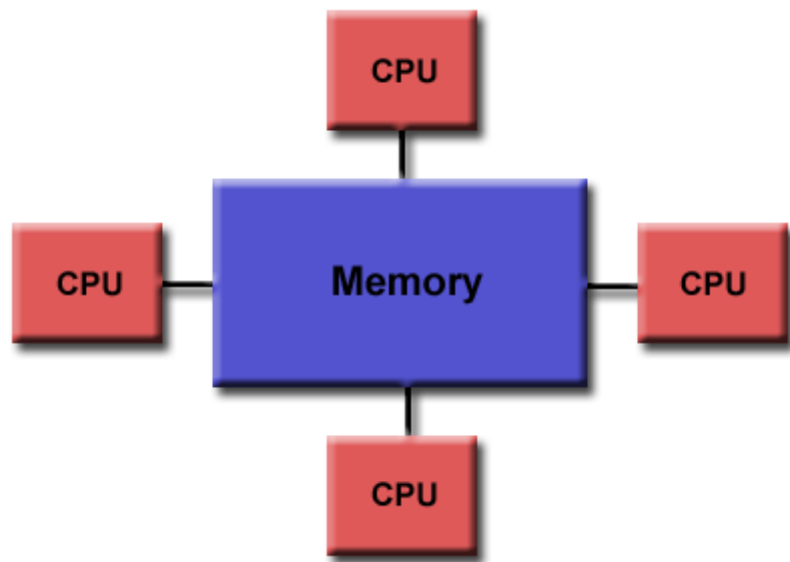
A classificação mais fundamental para as arquiteturas de paralelização está no compartilhamento de recursos.

Em alguns casos, sistemas de arquivos (discos, etc.) podem ser compartilhados: todas as unidades têm acesso aos mesmos arquivos.

A memória (RAM) normalmente é o mais importante recurso:

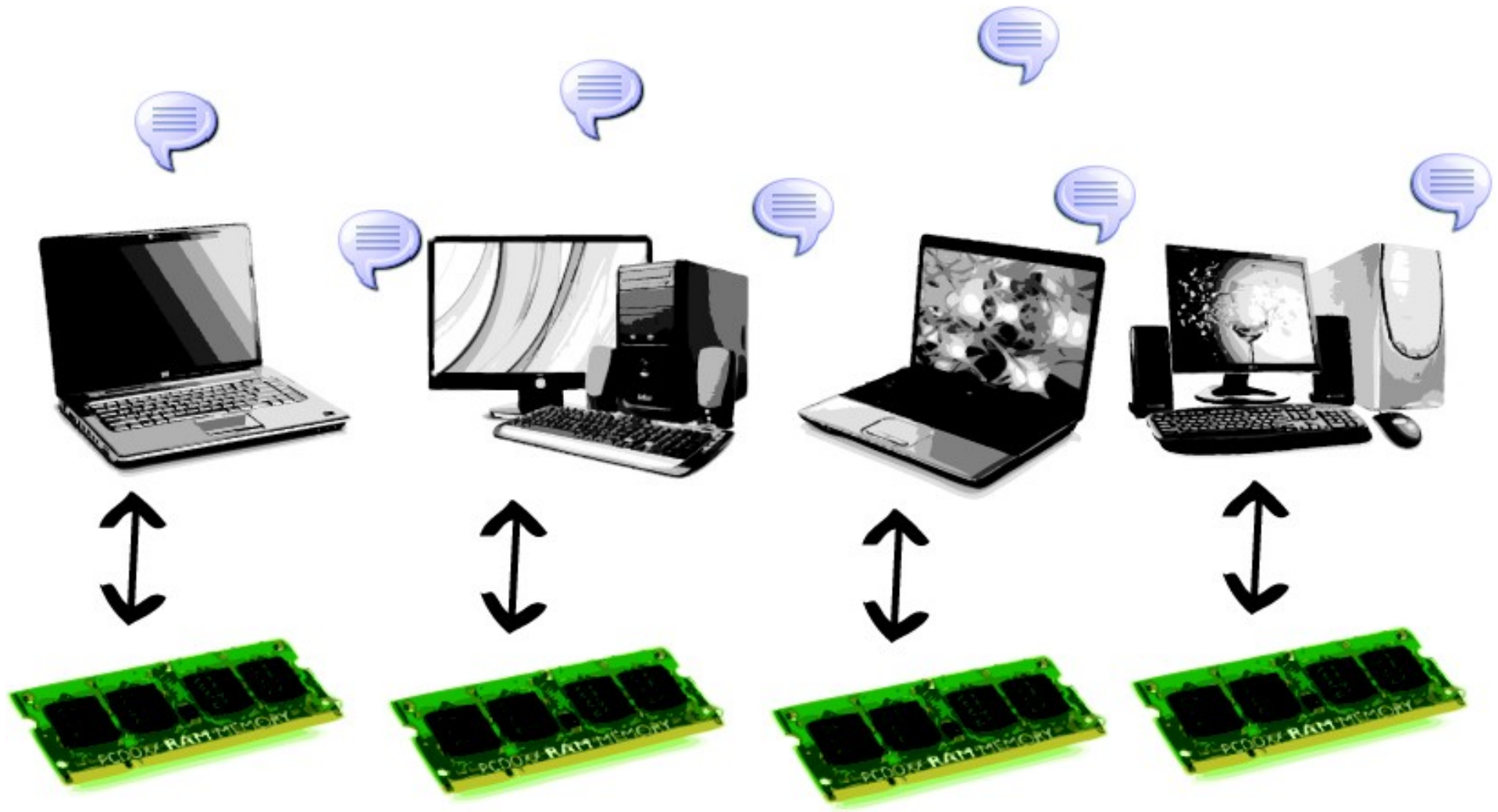
- É muito mais rápida (ordens de magnitude) que discos.
- Normalmente é onde ficam todas as variáveis que um programa usa.
- É muito mais cara (por isso mais escassa) que discos.
- Só raros (e caros) sistemas conseguem compartilhar memória entre nós (computadores) diferentes. Mesmo quando é possível, o acesso à memória entre nós é mais lento que dentro de um nó (por limitações físicas).

Quando há compartilhamento, **o mesmo recurso está disponível a todas as unidades de execução**. Todas podem (a princípio) acessar os mesmos dados, ao mesmo tempo:



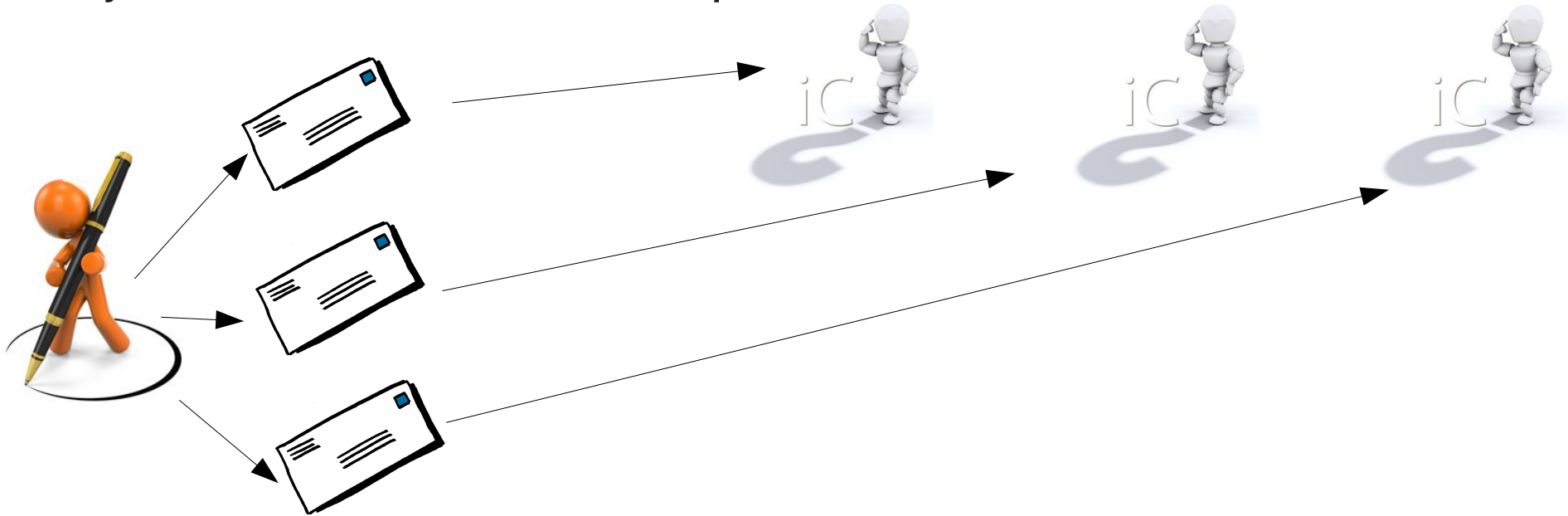
Principais arquiteturas paralelas atuais

Sem compartilhamento, **cada unidade tem uma cópia** dos recursos necessários a todas. Interação entre unidades se dá em só por troca de mensagens:



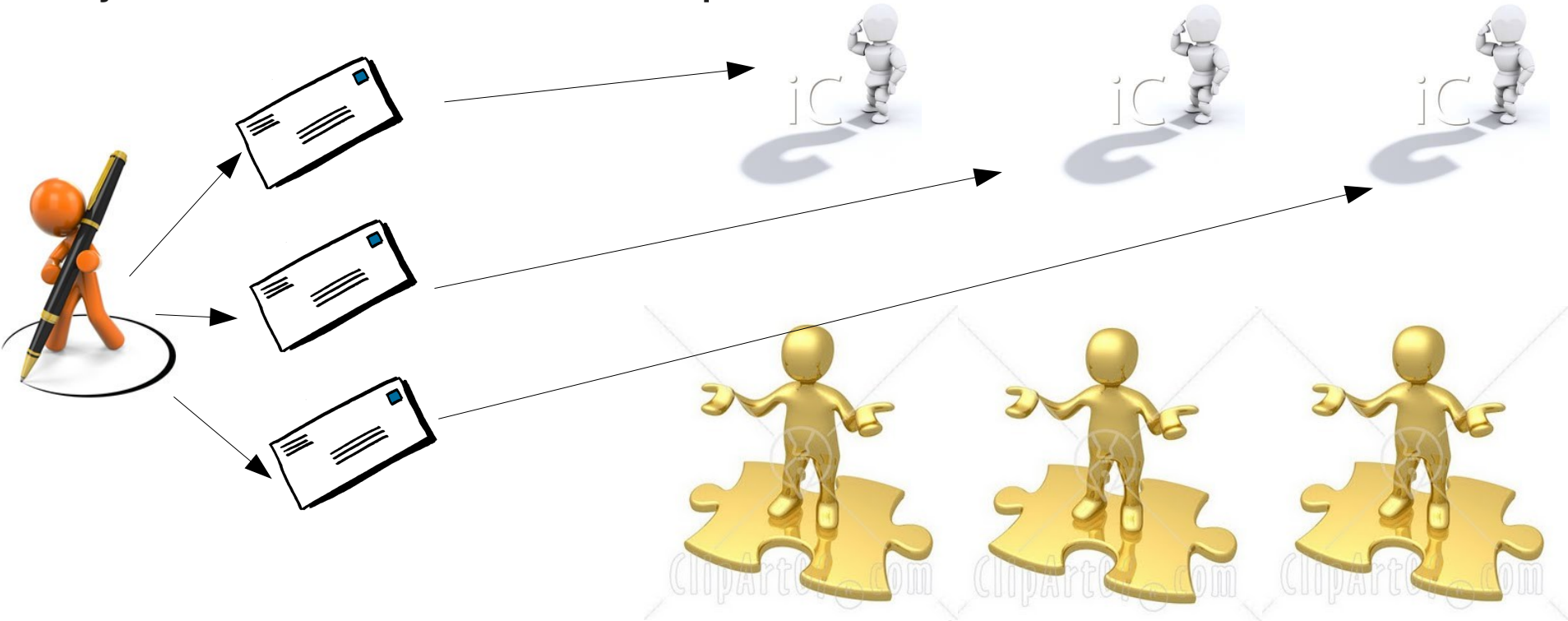
Principais arquiteturas paralelas atuais

Sem compartilhamento, **cada unidade tem uma cópia** dos recursos necessários a todas. **Interação entre unidades se dá em só por troca de mensagens:**



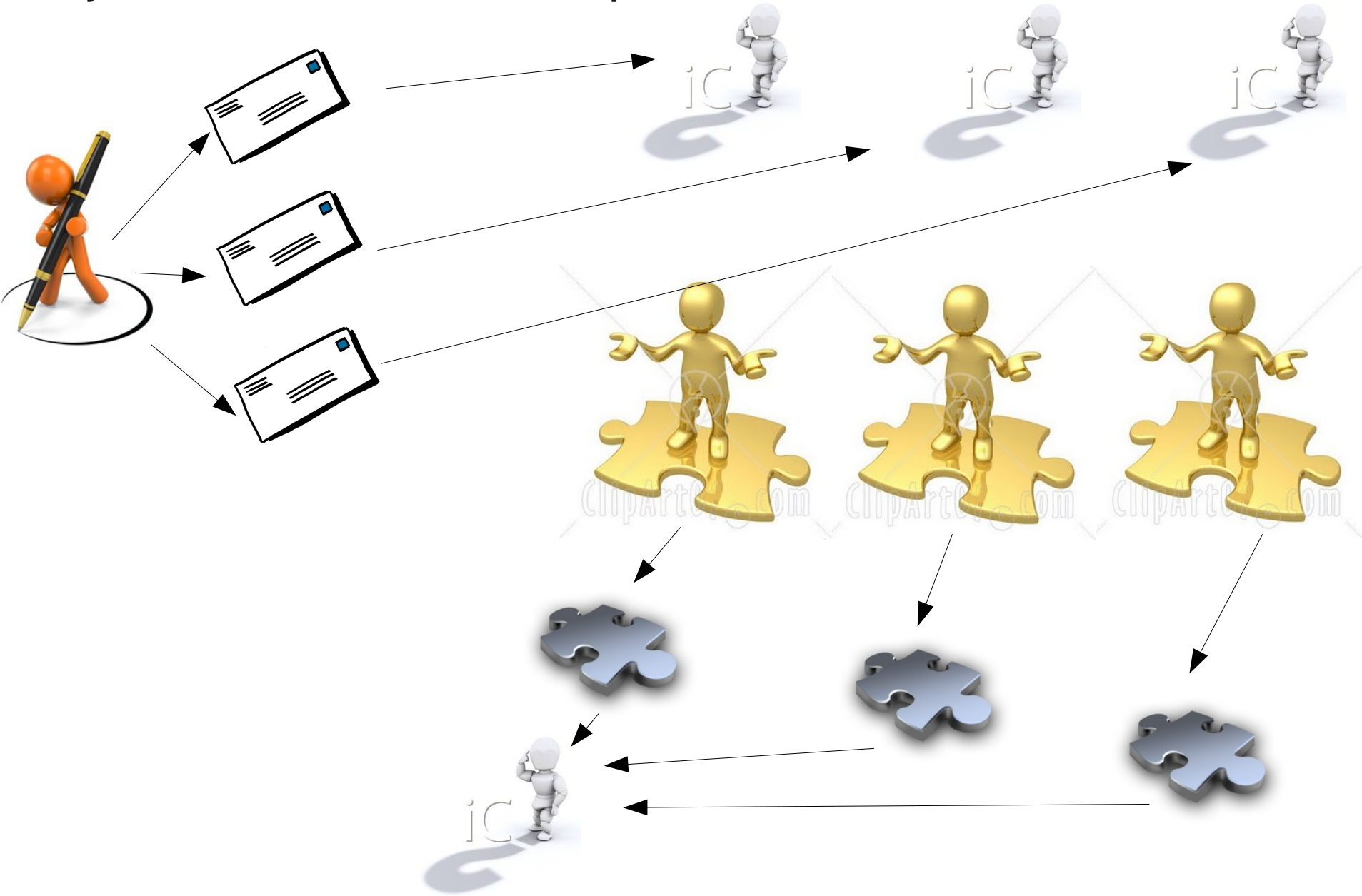
Principais arquiteturas paralelas atuais

Sem compartilhamento, **cada unidade tem uma cópia** dos recursos necessários a todas. **Interação entre unidades se dá em só por troca de mensagens:**



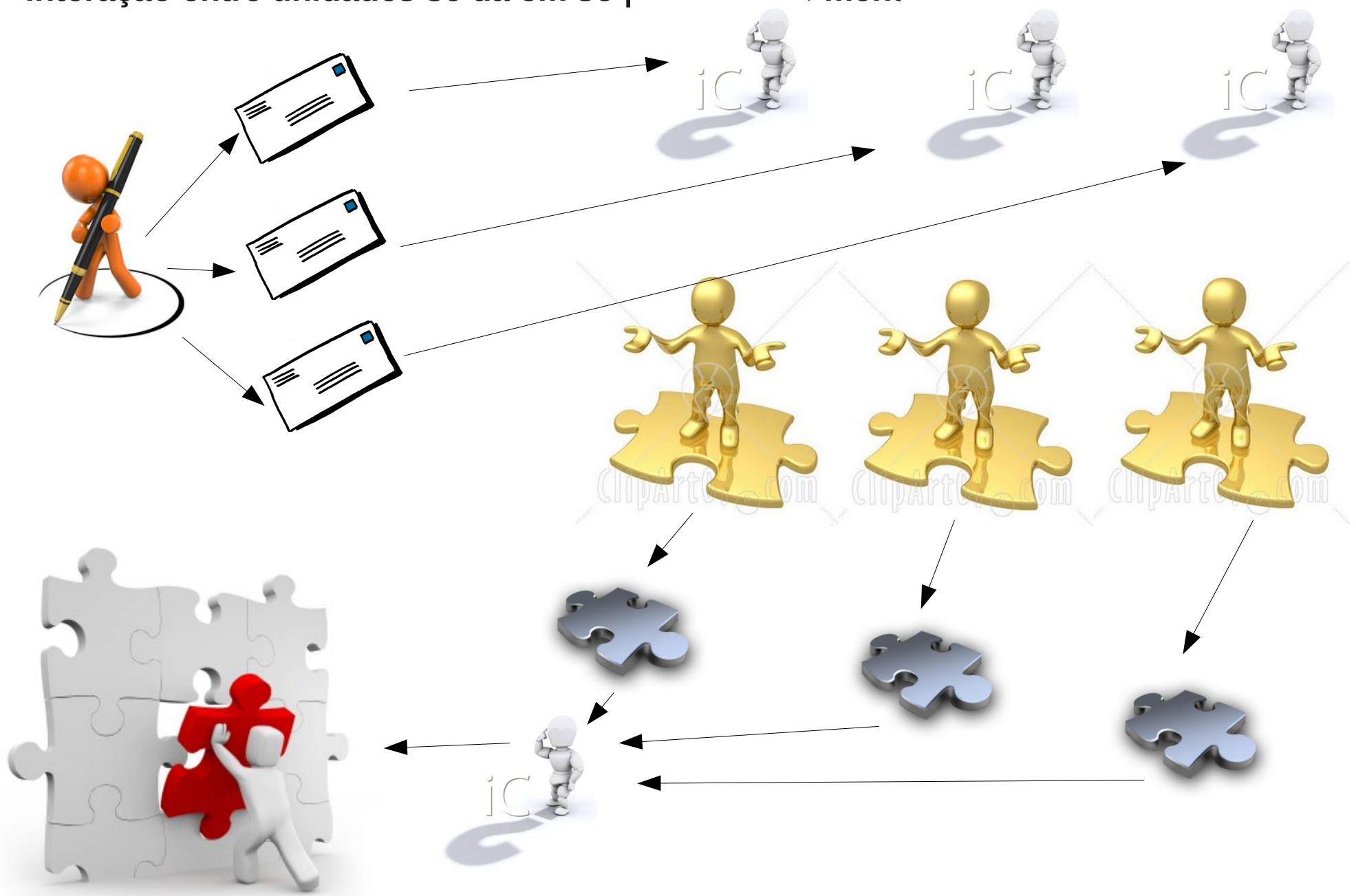
Principais arquiteturas paralelas atuais

Sem compartilhamento, **cada unidade tem uma cópia** dos recursos necessários a todas. Interação entre unidades se dá em só por troca de mensagens:



Principais arquiteturas paralelas atuais

Sem compartilhamento, **cada unidade tem uma cópia** dos recursos necessários a todas. Interação entre unidades se dá em só por troca de mensagens:



Principais arquiteturas paralelas atuais

Com memória compartilhada*:

Cada unidade de execução é um *thread*.

- Vetorização (adiante)
- OpenMP (próxima aula)
- GPUs (em outro curso)
- Gerenciamento manual de *threads* (dependente de linguagem e sistema operacional)

Sem memória compartilhada*:

Cada unidade de execução é um *processo*.

- MPI (próxima aula)
- Grids, clouds (outro curso)
- Gerenciamento manual de *processos* (dependente de linguagem e sistema operacional)

*Mesmo sem memória compartilhada é possível (e comum) compartilhar arquivos.

Paradigmas discutidos neste curso

Vetorização

- **Operações sobre arrays** são expressas com a semântica da linguagem.
- O programador não especifica como dividir o trabalho entre as unidades, as coordenar ou juntar os resultados ao final.
- **A paralelização é feita pelo compilador / interpretador.**
- Traz vantagens (de organização de código e eficiência) mesmo se não houver paralelização.

OpenMP

- **Usa memória compartilhada.**
- O mais comum e melhor padronizado sistema para paralelismo de dados.
- Pode ser usado para paralelismo de tarefas.
- Simples de usar, inclusive paralelizar gradualmente.
- Muito mais simples que gerenciar threads e sua comunicação diretamente.

MPI

- **Usa processos (memória distribuída)**, mesmo que dentro do mesmo computador.
- O sistema mais comum para uso de clusters (vários computadores interligados) e paralelismo de tarefas.
- Em geral o mais complexo de usar destes 3.
- Muito mais simples que gerenciar processos e sua comunicação diretamente.

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:

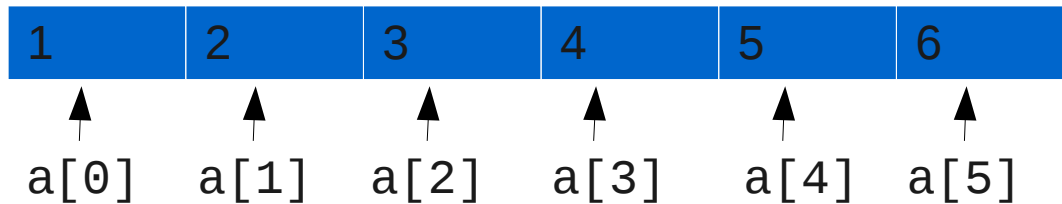
- 1D é muito comum.
- MD é com frequência desajeitado (2D ocasionalmente não é tão ruim): **IDL e Python+Numpy têm arrays MD de muito alto nível** (adiante).

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.

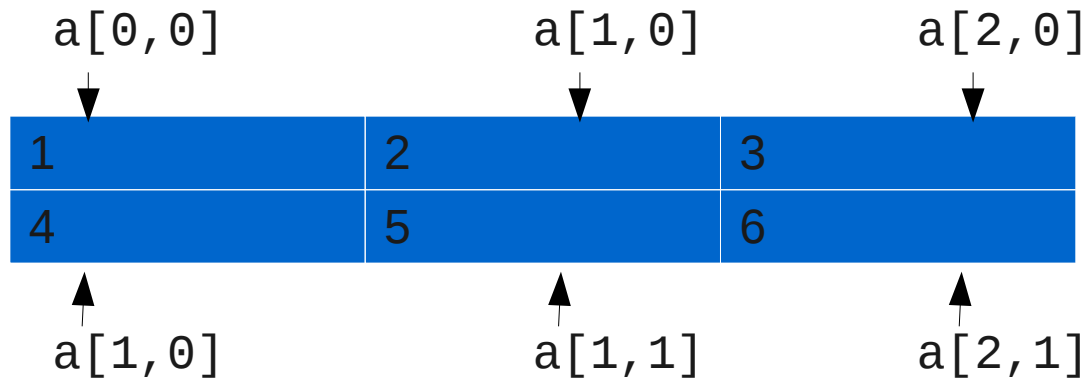
IDL> `help, a`
A **INT** **= Array[6]**

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`

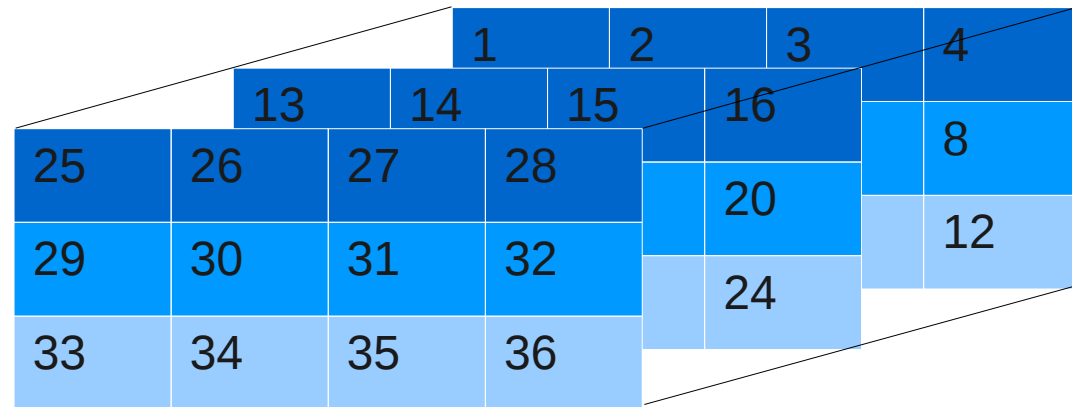
```

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

```



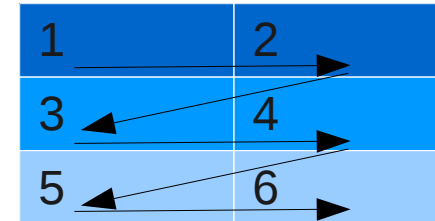
Para mais que 3D, a imagem gráfica costuma ser conjuntos de pilhas 3D (para 4D), conjuntos de 4D (para 5D), etc.

Arrays - armazenamento MD

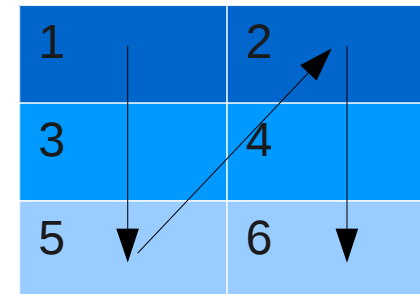
Se internamente são sempre seqüências 1D, como são armazenados arrays MD?

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
```

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

```
IDL> c=b[1:2,2:4]
```

```
IDL> help,c
```

```
C          BYTE          = Array[2, 3]
```

```
IDL> print,c
```

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

→ **Todas** as colunas, linhas **0 a 2**

```
IDL> print,b[*,0:2]
```

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

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

```
IDL> print,b[1:2,0:-1:2]
```

```
 1  2
 9 10
17 18
```

→ O passo pode ser negativo, para andar na ordem reversa

```
IDL> print,b[1,2:0:-1]
```

```
 9
 5
 1
```

*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
```

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

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

`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) ← 1D, 4 elementos
IDL> help,e
E                BYTE      = Array[4]
IDL> print,e
  0   1   2   3
IDL> f=bindgen(1,4) ← 2D, 4 elementos: 1 coluna, 4 linhas
IDL> help,f
F                BYTE      = Array[1, 4]
IDL> print,f
  0
  1
  2
  3
IDL> g=bindgen(4,1) ← 2D, 4 elementos: 4 colunas, 1 linha: equivalente a 1D,
IDL> help,g
G                BYTE      = Array[4]
IDL> print,g
  0   1   2   3

```

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
```

Concatenamento na primeira dimensão (a da esquerda)

```
IDL> d=[[a], [b]]
IDL> help, d
```

```
D          INT          = Array[3, 2]
IDL> print, d
      1      2      3
      4      5      6
```

Concatenamento na segunda dimensão

```
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 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):

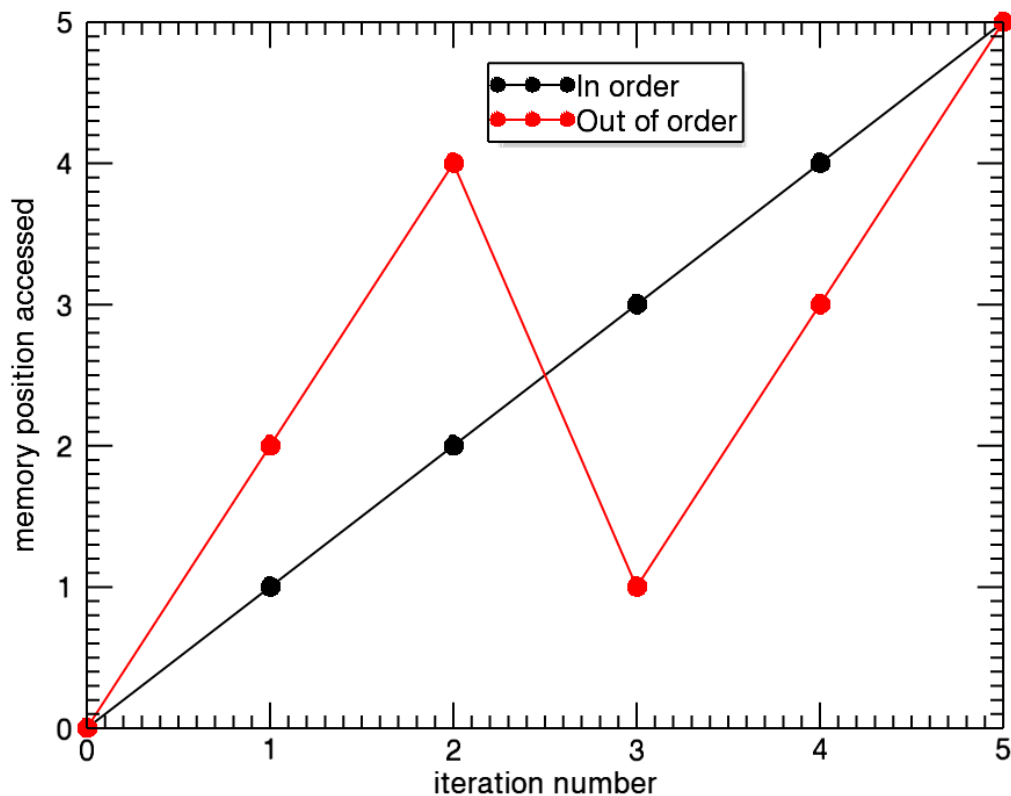
```

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):

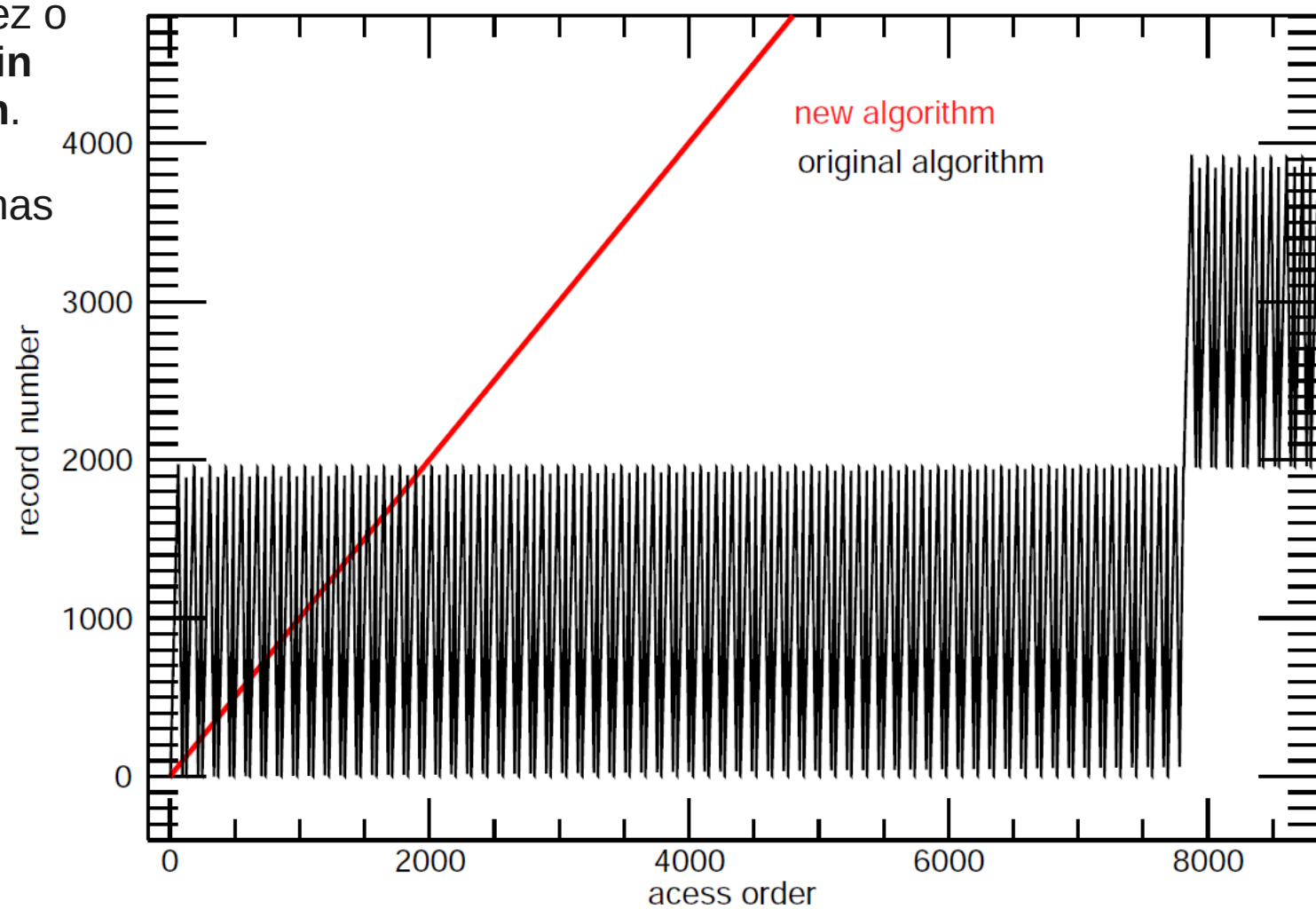


Arrays - armazenamento MD - exemplos

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:

$$\mathbf{c} = \mathbf{a} + \mathbf{b}$$

$$\mathbf{d} = \sin(\mathbf{c})$$

Onde \mathbf{a} e \mathbf{b} são arrays de qualquer dimensão (\mathbf{a} e \mathbf{b} têm as mesmas dimensões)

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

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

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 (em graus progressivos), C++, Java: vetorização simples:
 - desajeitada para mais de 1D (pior ainda para mais de 2D), limitada a operações sobre arrays inteiros ou fatias regulares (mostrada adiante)
- 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 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]] → Array 1D, 4 elementos: 0,1,3,5 de a
      0          2          6          10
IDL> print, [[0,1],[3,5]] → Array 2D de índices (1D), 4 elementos
      0          1
      3          5
IDL> print, a[[[0,1],[3,5]]] → Array 2D, 4 elementos de a, dados pelos
      0          2          índices (1D) acima.
      6          10
```

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]] → Array 1D, 2 elementos:
      16          18          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 conformes 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).

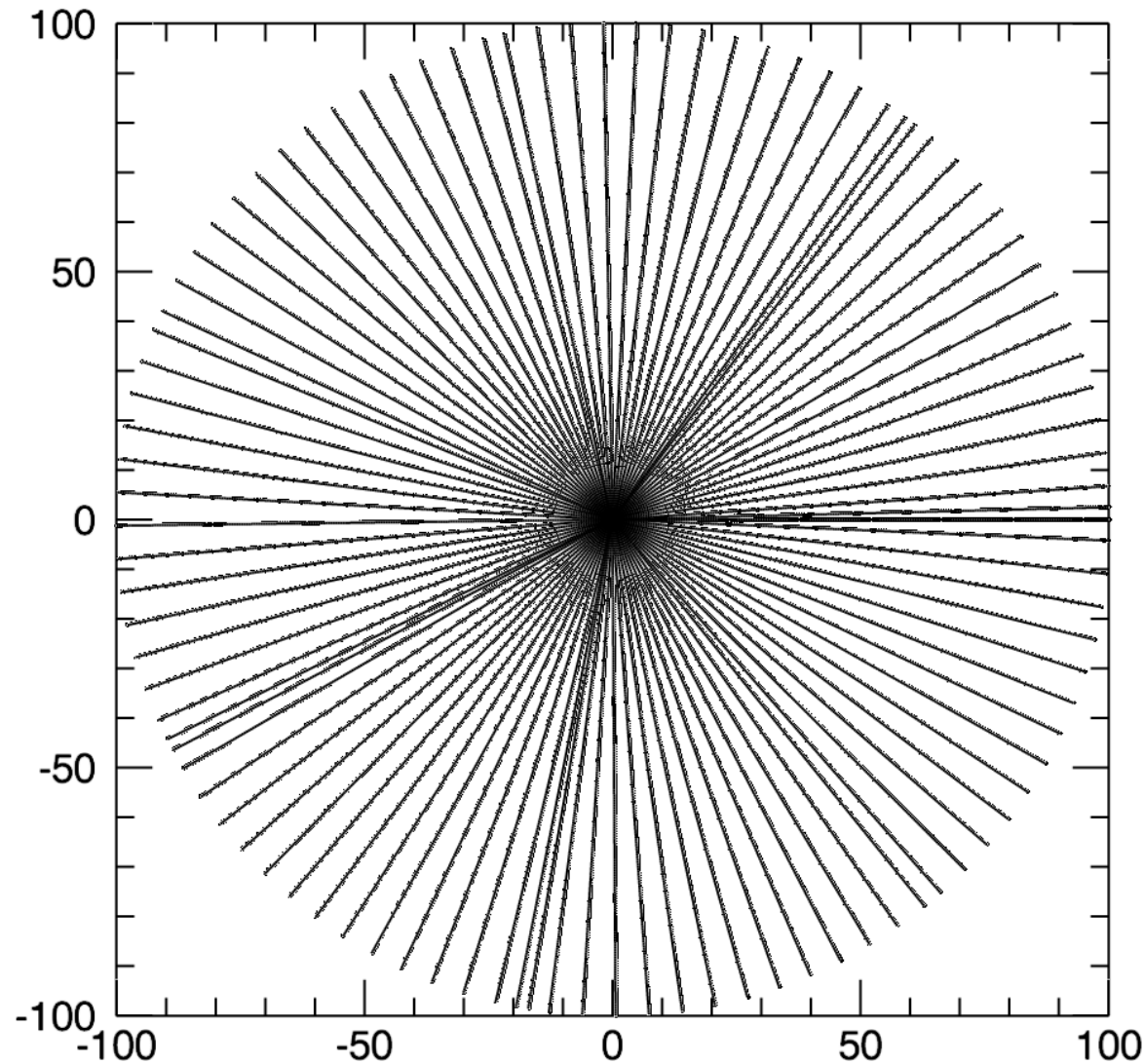
R, os elementos do array menor são repetidos o suficiente, ciclicamente:

```
> 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 , θ) 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
  0.10000000    0.60251256    1.1050251
  0.10000000    0.60251256    1.1050251
```

```
  1.6075377
  1.6075377
  1.6075377
```

```
IDL> print, theta_2d[0:3, 0:2]
```

```
  0.00000000    0.00000000    0.00000000
  3.60000000    3.60000000    3.60000000
  7.20000000    7.20000000    7.20000000
```

```
  0.00000000
  3.60000000
  7.20000000
```

```
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   12.0000000
 12.0000000   16.0000000
```

→ Cada linha tem as coordenadas de um ponto

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

```
IDL> r=sqrt(total(xy^2,1))
IDL> print,r
```

```
  5.0000000    10.0000000    15.0000000    20.0000000
```

Dimensão sobre a qual é realizada a soma.

O mesmo algoritmo 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   10.0000000   11.0000000
```

```
IDL> r=sqrt(total(xyz^2,1))
IDL> print,r
```

```
  2.2360680    7.0710678    12.206556    17.378147
```

```
IDL> print,r^2
```

```
  5.0000000    50.0000000   149.000000   302.000000
```

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 $[3, ns, n1]$ para $[3, ns*n1]$, 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 $H\alpha$:

```
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      100.000000
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  s1_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      's1_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 algoritmos 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=r1,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://ppentado.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,r1
RL          LIST <ID=44  NELEMENTS=14>
IDL> foreach element,locations,i do print,element,' : ',r1[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 **r1** 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[r1[6]]
2455569
{ HD59382      2455569 s1_0102.fits<ObjHeapVar8(HASH)>}
{ HD63281      2455569 s1_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
```

2

0

2

1

3

1

4

2

2

2

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=r1,reverse_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 e1,r1 do oindexes.add,indexes[e1]
IDL> olocations=updates[locations]
IDL> foreach e1,r1,i do print,olocations[i], ' : ',obs[e1].object
2455563 : HD28985
2455569 : HD59382 HD63281
2455570 : HD63281 HD48561 HD78325 HR892561 HR748267
2455574 : HR189365
2455576 : HR167382
```

Algumas referências

Vetorização

- *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/>
- 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 e vetorização

- 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
- Vetorização
- Algumas referências

Slides em http://www.ppenteadonet/ast/pp_para_1.pdf

2 – OpenMP, MPI, escolhas

- OpenMP
- MPI
- Escolha de forma de paralelização
- Algumas referências

Slides em http://www.ppenteadonet/ast/pp_para_2.pdf

Exemplos em http://www.ppenteadonet/ast/pp_para/

pp.penteadon@gmail.com