Paralelização de códigos, OpenMP, MPI e GPUs

1 – Introdução

Paulo Penteado
Northern Arizona University

pp.penteado@gmail.com

Esta apresentação: http://www.ppenteado.net/ast/pp_para_iiwcca_1.pdf
Arquivos do curso: http://www.ppenteado.net/ast/pp_para_iiwcca/

Programa

1 – Introdução

- Motivação
- Formas de paralelização
  - Paralelismo de dados
  - Paralelismo de tarefas
- Principais arquiteturas paralelas atuais
  - Com recursos compartilhados
  - Independentes
- Comparação de paradigmas discutidos neste curso:
  - Vetorização
  - OpenMP
  - MPI
  - GPUs
  - Paralelização extrínseca
  - Paralelização indireta
- Vetorização
- Paralelização extrínseca
- Algumas referências

Slides em http://www.ppenteado.net/ast/pp_para_iiwcca_1.pdf
Exemplos em http://www.ppenteado.net/ast/pp_para_iiwcca/

pp.penteado@gmail.com

Motivação

Como tornar programas mais rápidos?

Motivação

Como tornar programas mais rápidos?

É só esperar ter computadores mais rápidos?

Motivação

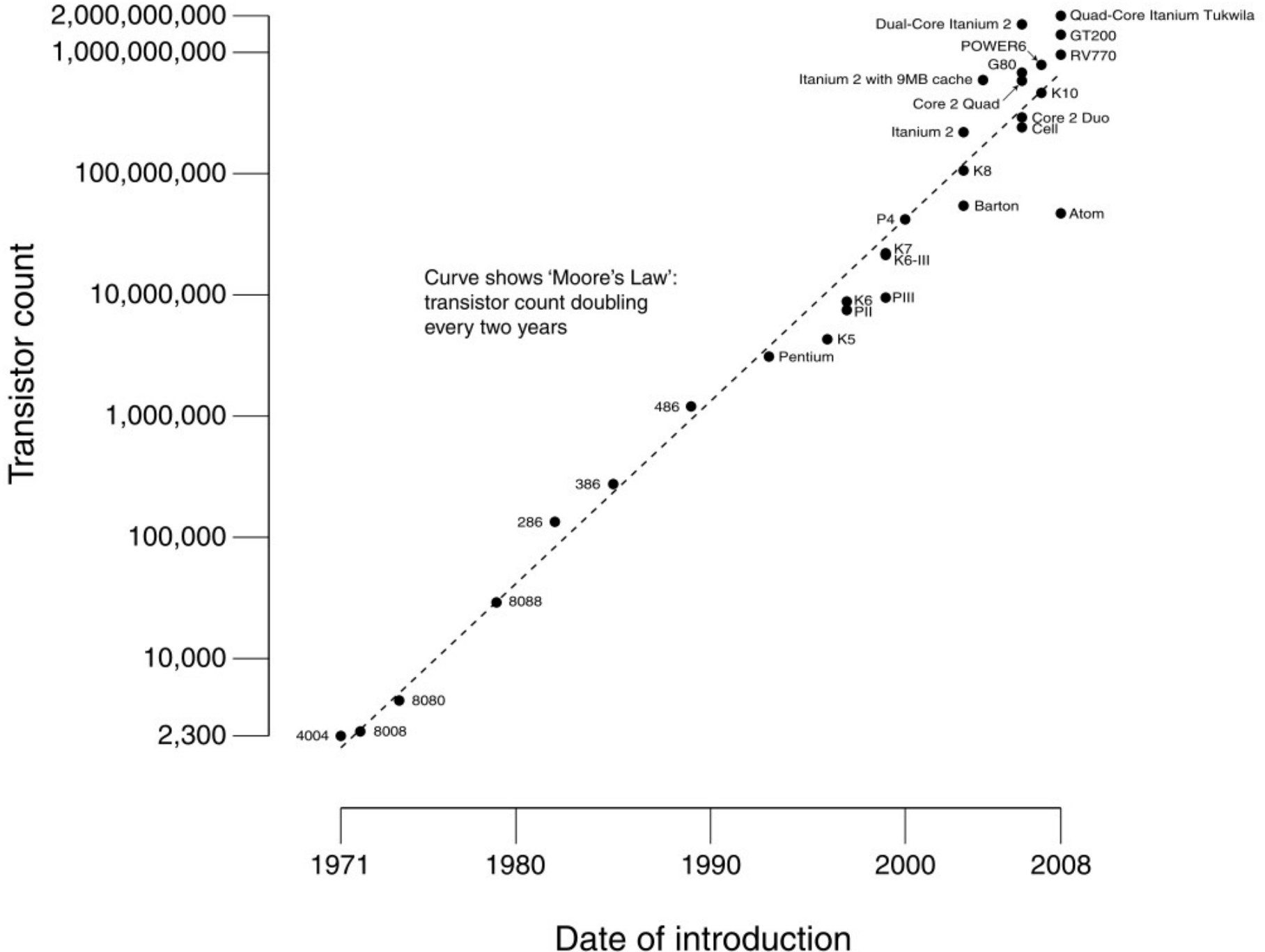
Como tornar programas mais rápidos?

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.

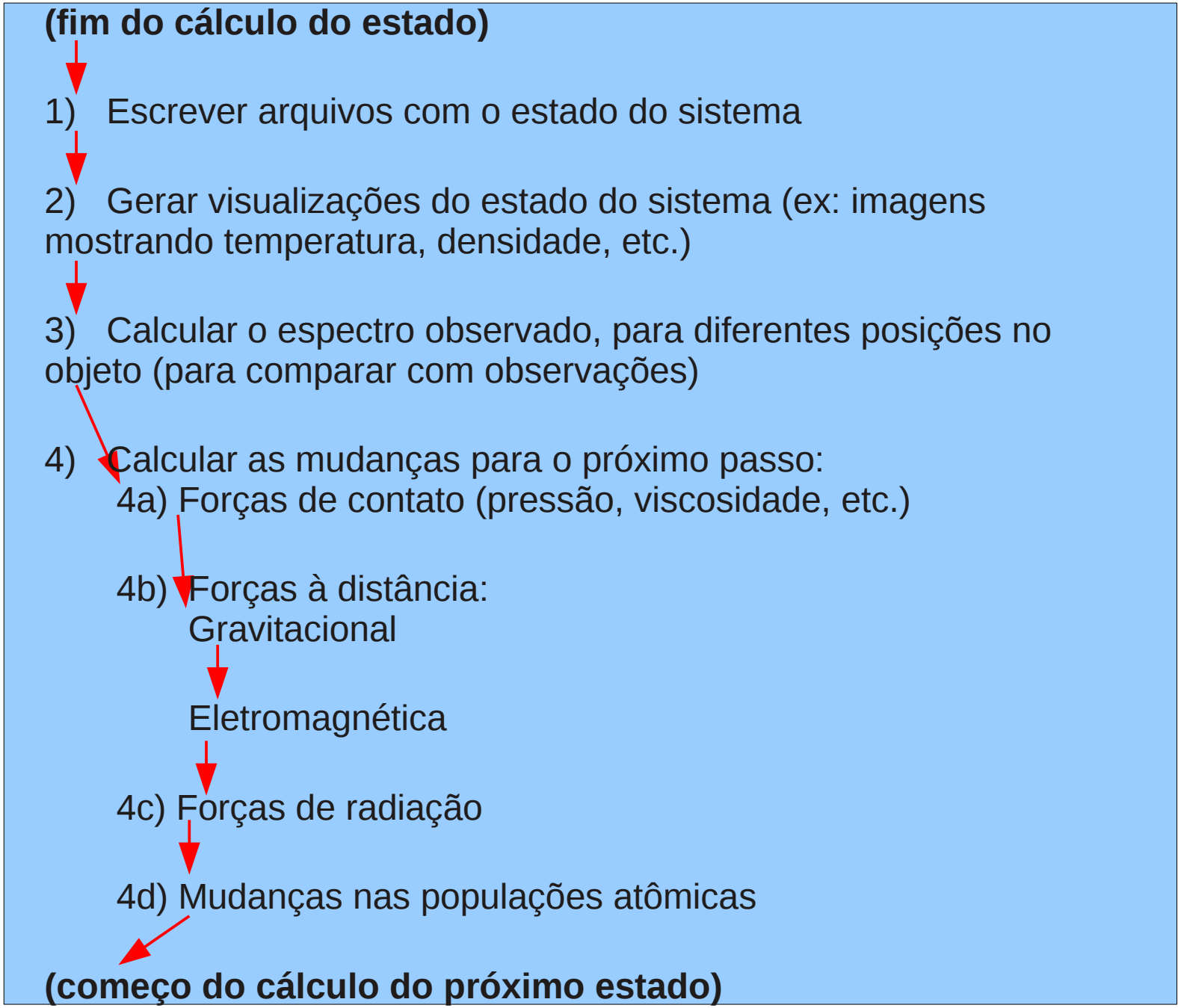
- Tarefas que contém partes calculadas independentemente:
  - Operações vetoriais (operações sobre arrays, álgebra linear, etc.)
  - Processamento de cada pixel de uma imagem.
  - Cada comprimento de onda em um modelo de transferência radiativa.
  - Cada célula / partícula de um modelo dinâmico.

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:

Código serial: uma coisa feita de cada vez

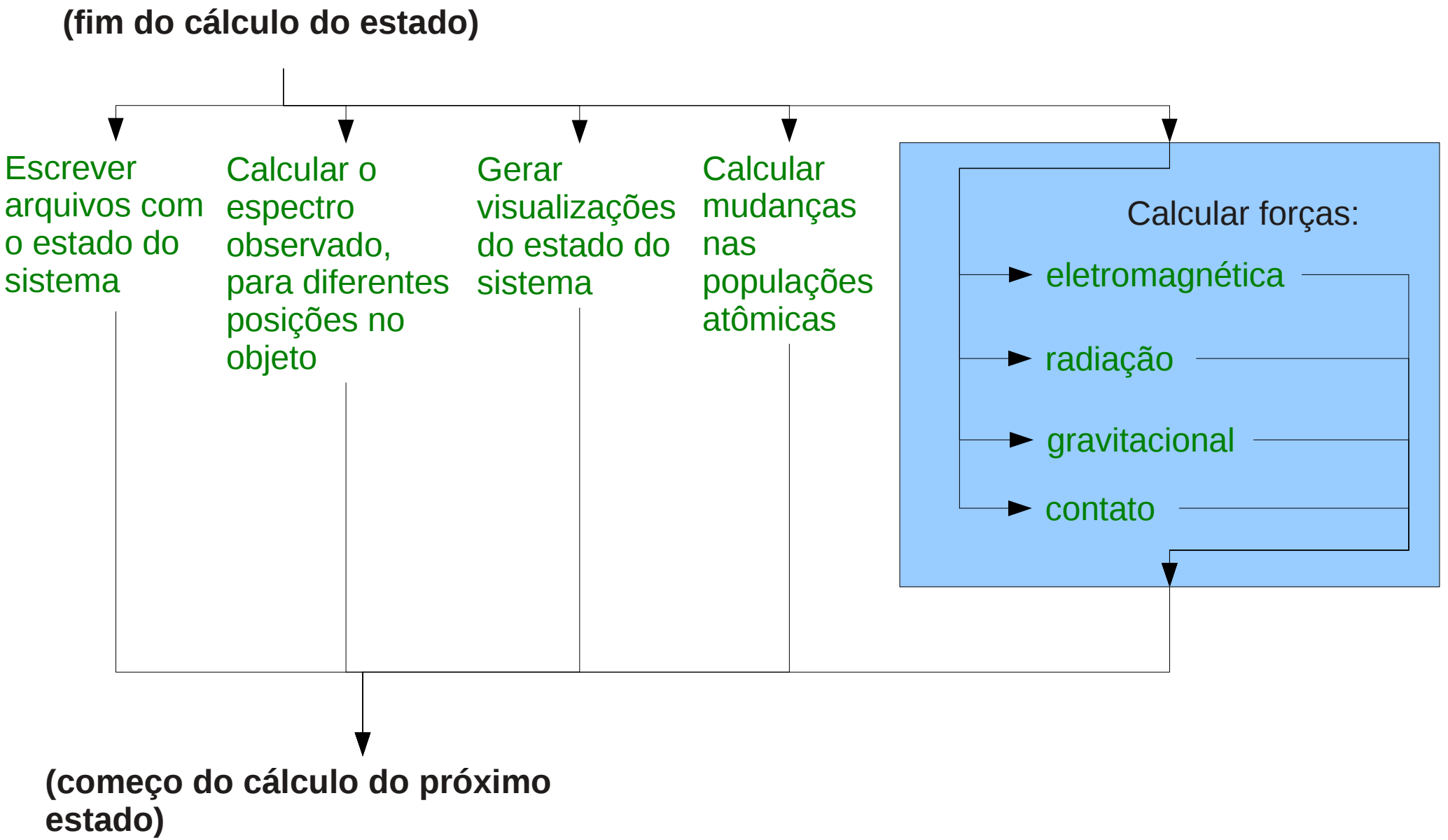
(cada ítem numerado sendo uma rotina chamada, cada uma após terminar a anterior)



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



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.

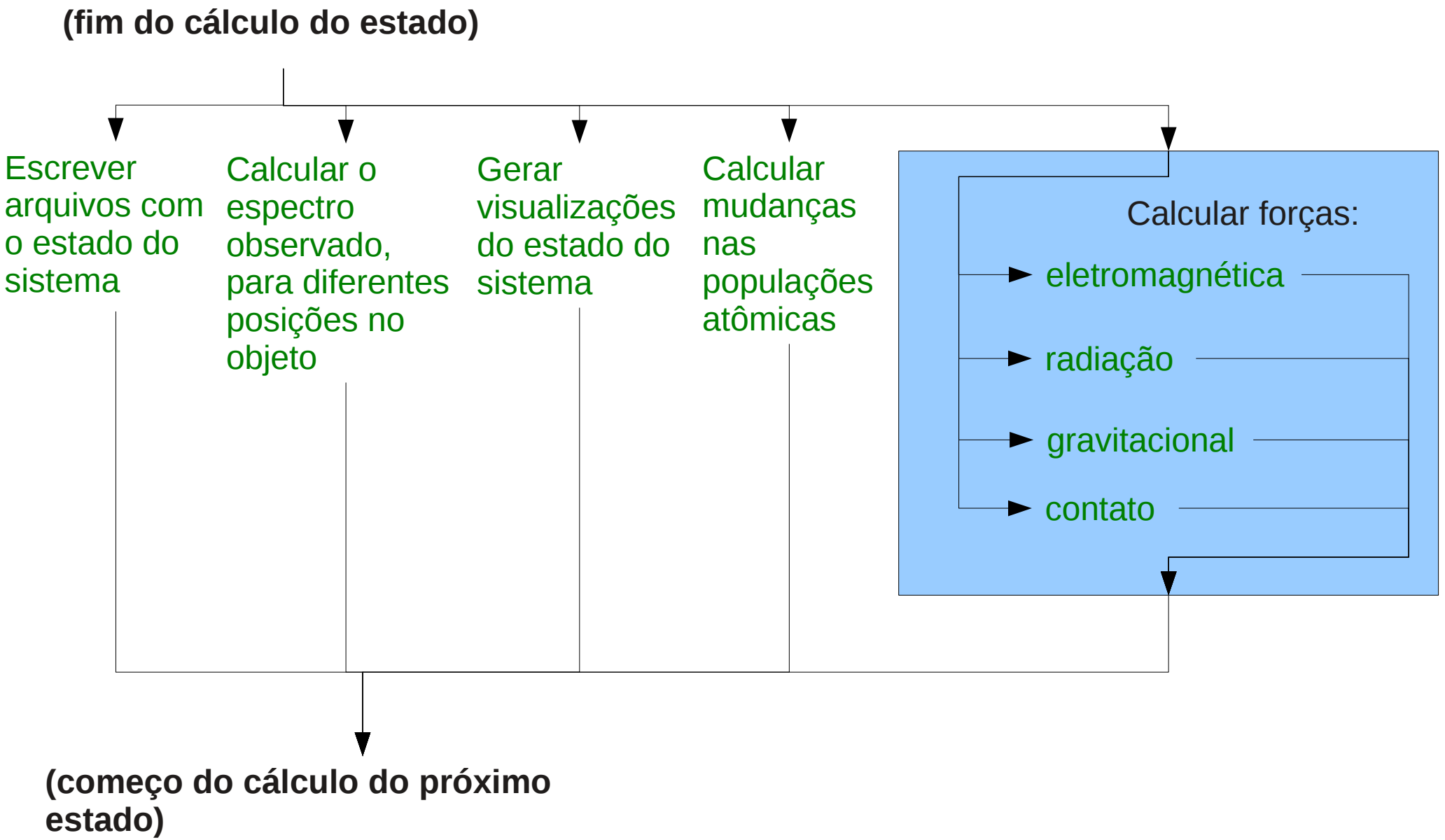
- Tarefas que contém partes calculadas independentemente:
  - Operações vetoriais (operações sobre arrays, álgebra linear, etc.)
  - Processamento de cada pixel de uma imagem.
  - Cada comprimento de onda em um modelo de transferência radiativa.
  - Cada célula / partícula de um modelo dinâmico.

*Este caso pode ser feito por **paralelização extrínseca - "*embarrassingly parallel problems*"**: 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:



Formas de paralelização - classificação

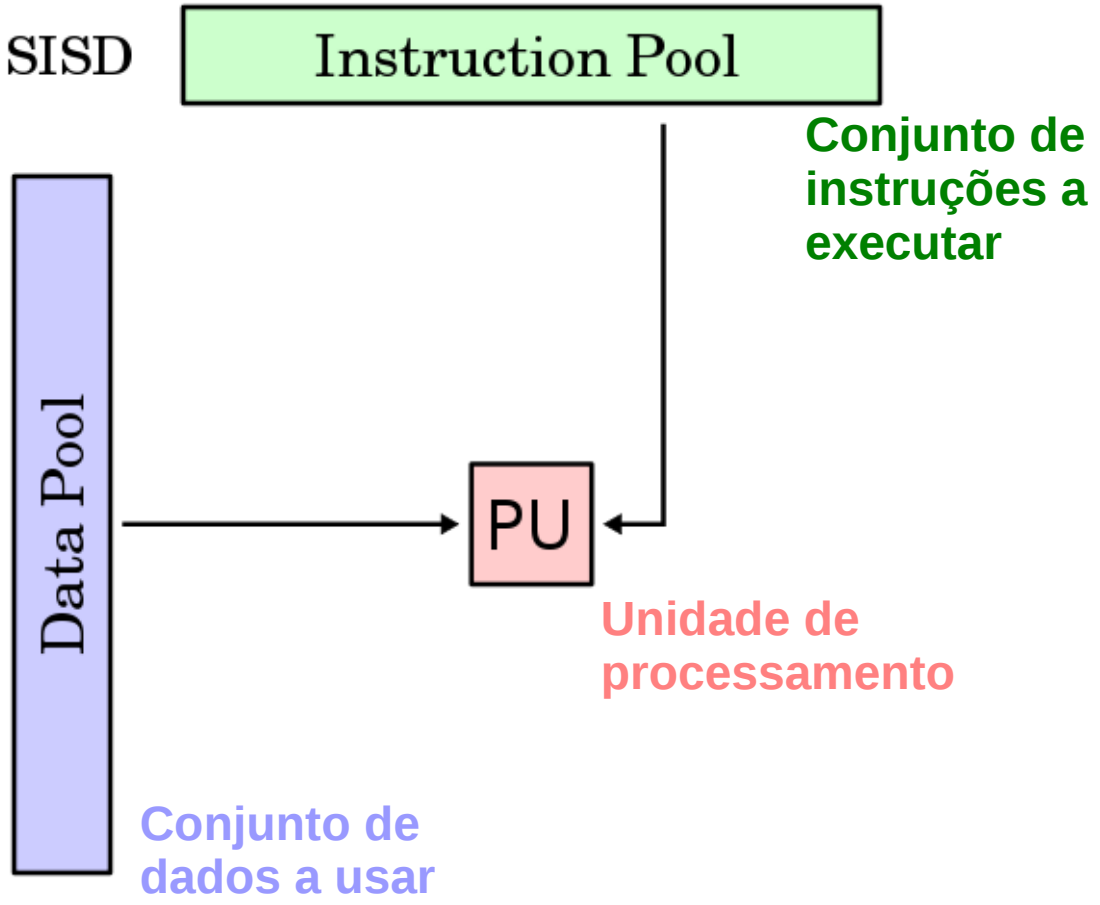
Taxonomia de Flynn (1966):

(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**

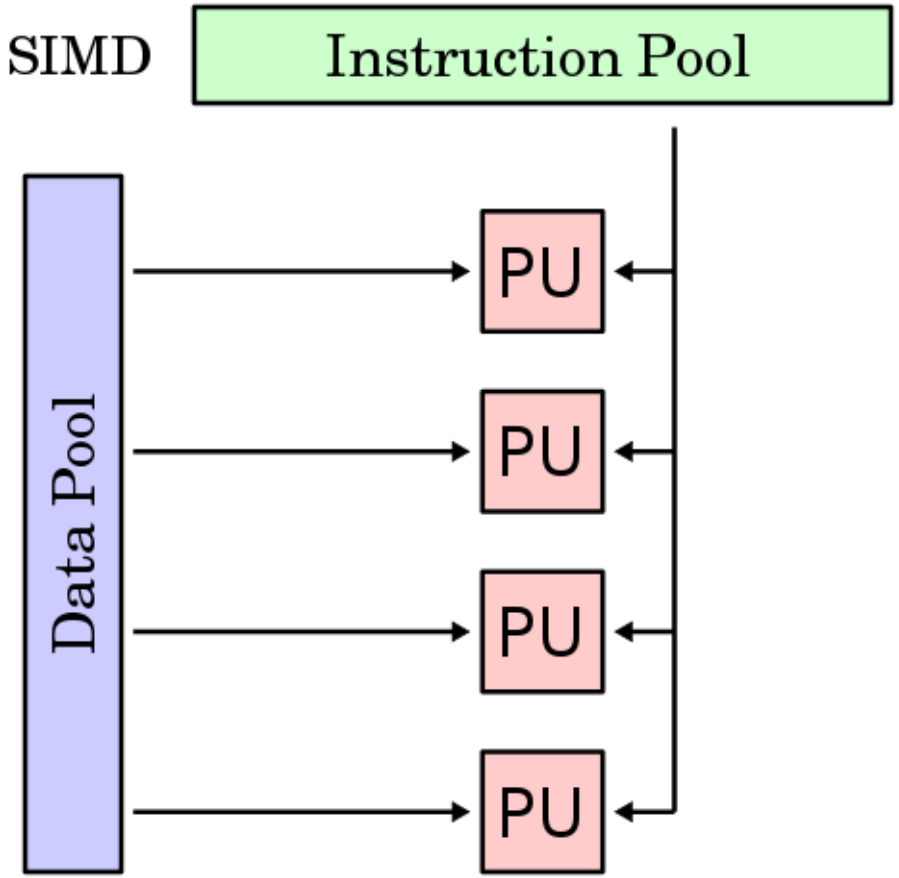
2) SIMD – Single Instruction, Multiple Data

Paralelismo de dados.

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

Exs:

- Operações vetorizadas
- Alguns usos de OpenMP e MPI
- GPUs



Formas de paralelização - classificação

Taxonomia de Flynn (1966):

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

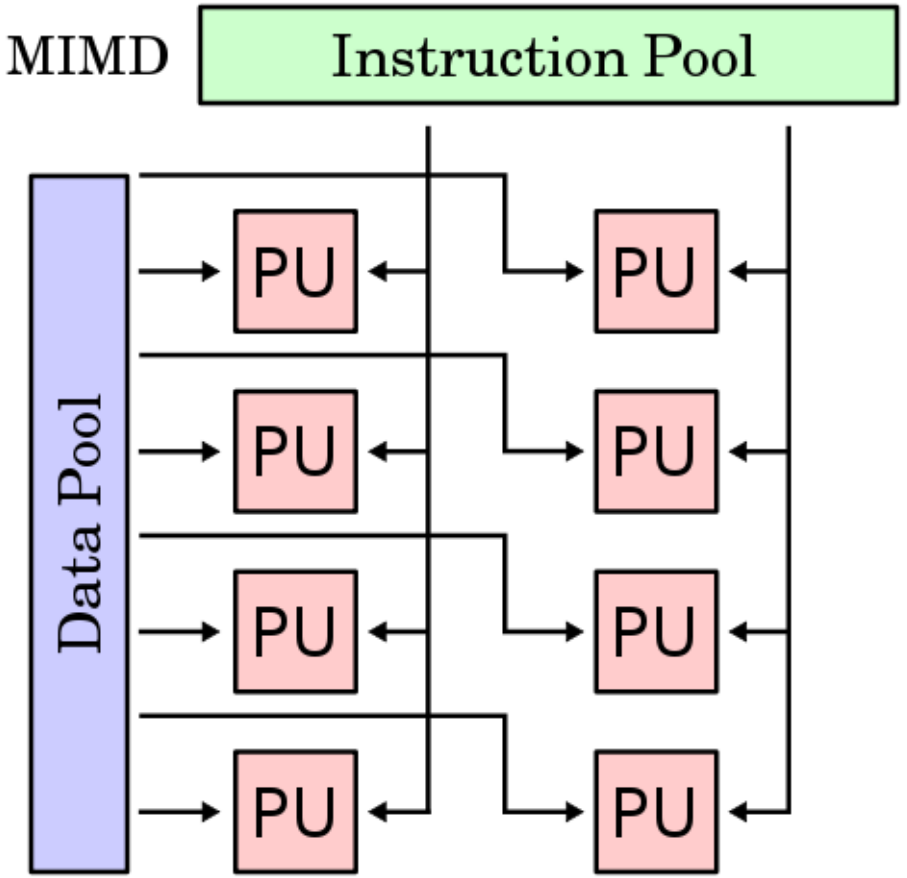
3) MIMD – Multiple Instruction, Multiple Data

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
- 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:

- *Não importa o número de padeiros, não vai levar menos tempo para um pão assar*.
- Mas é possível **N** padeiros, simultaneamente, produzirem um muitos pães: ainda leva o mesmo tempo 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 algorítimo:

- 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:

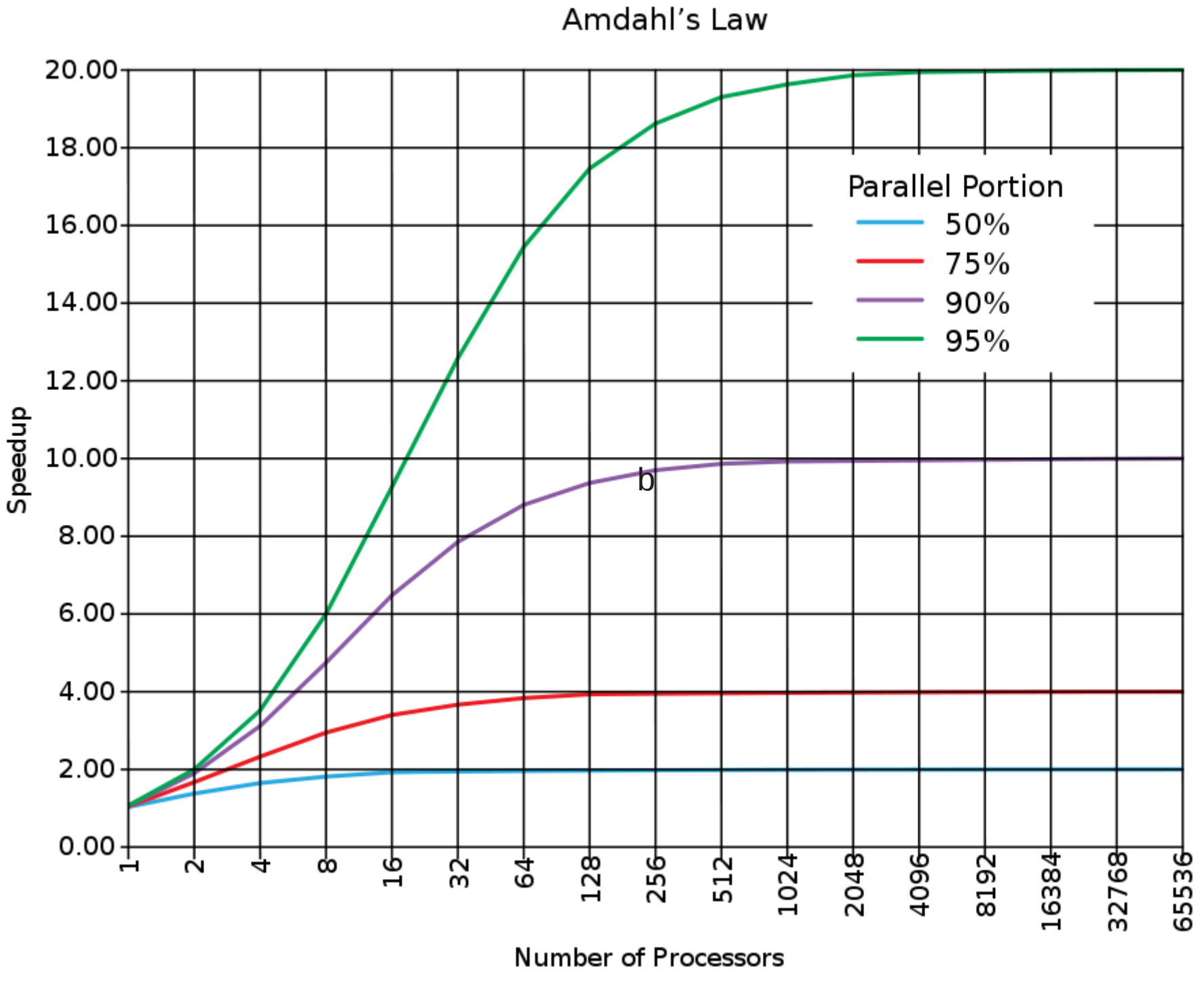
$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

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

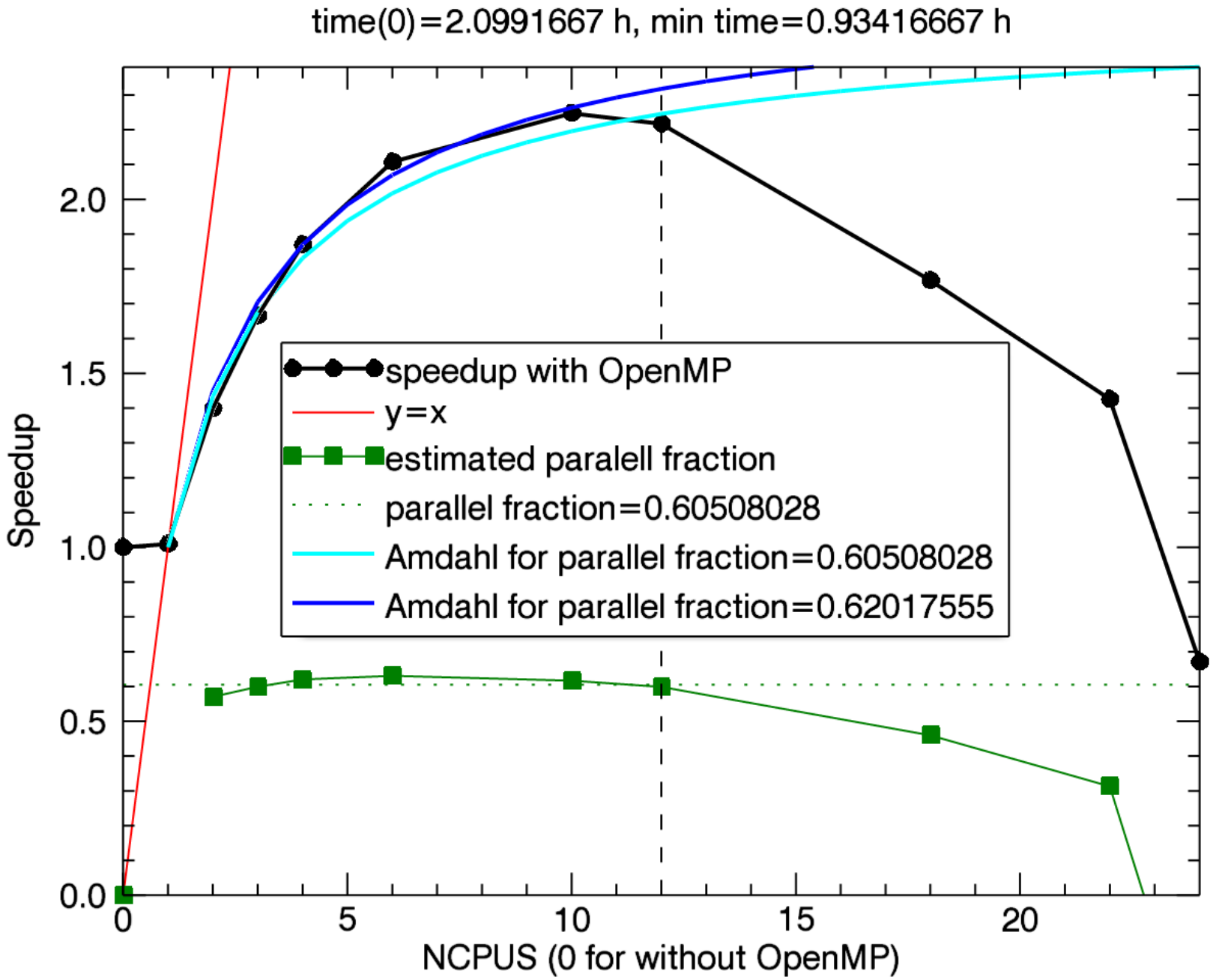
Não considera custos adicionais (*overheads*) de paralelização.

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

Lei de Amdahl – exemplos teóricos



Lei de Amdahl – exemplo real de medida



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

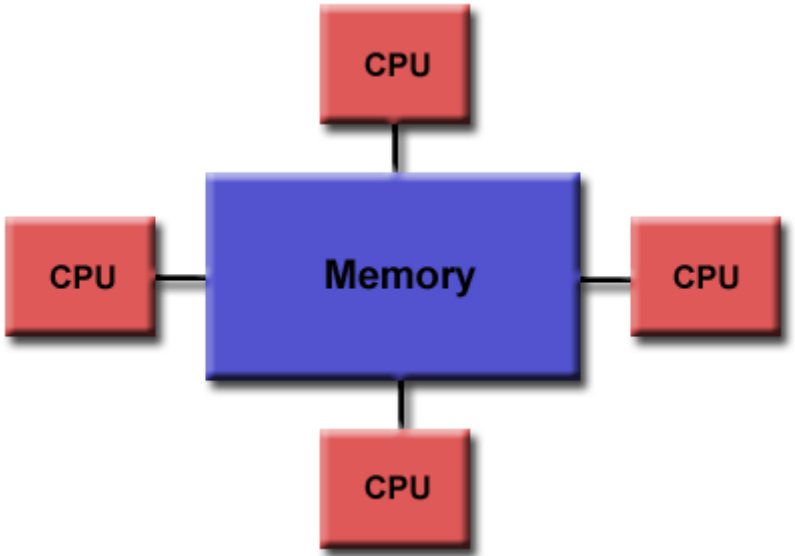
A classificação mais fundamental para as arquiteturas de paralelização está no compartilhamento de recursos.

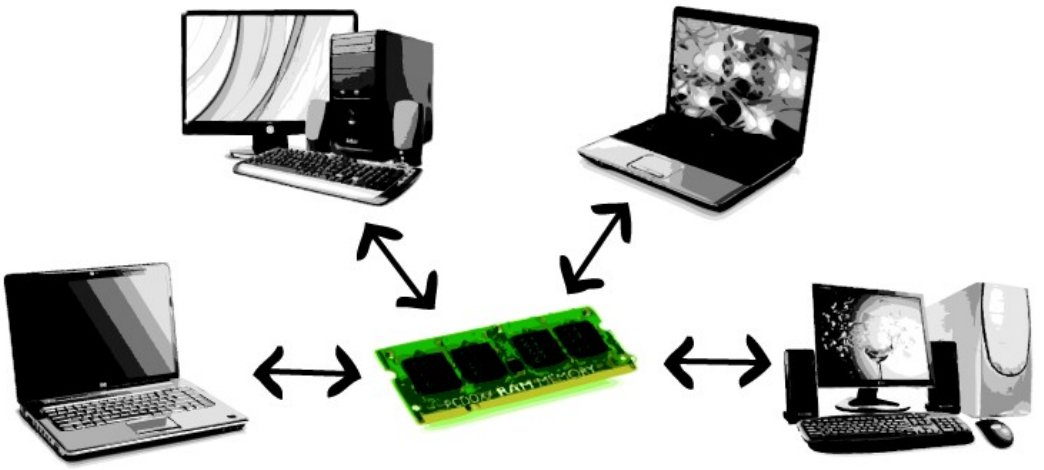
Em alguns casos, sistemas de arquivos (discos, etc.) podem ser compartilhados: todas as unidades têm acesso aos mesmos arquivos.

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).

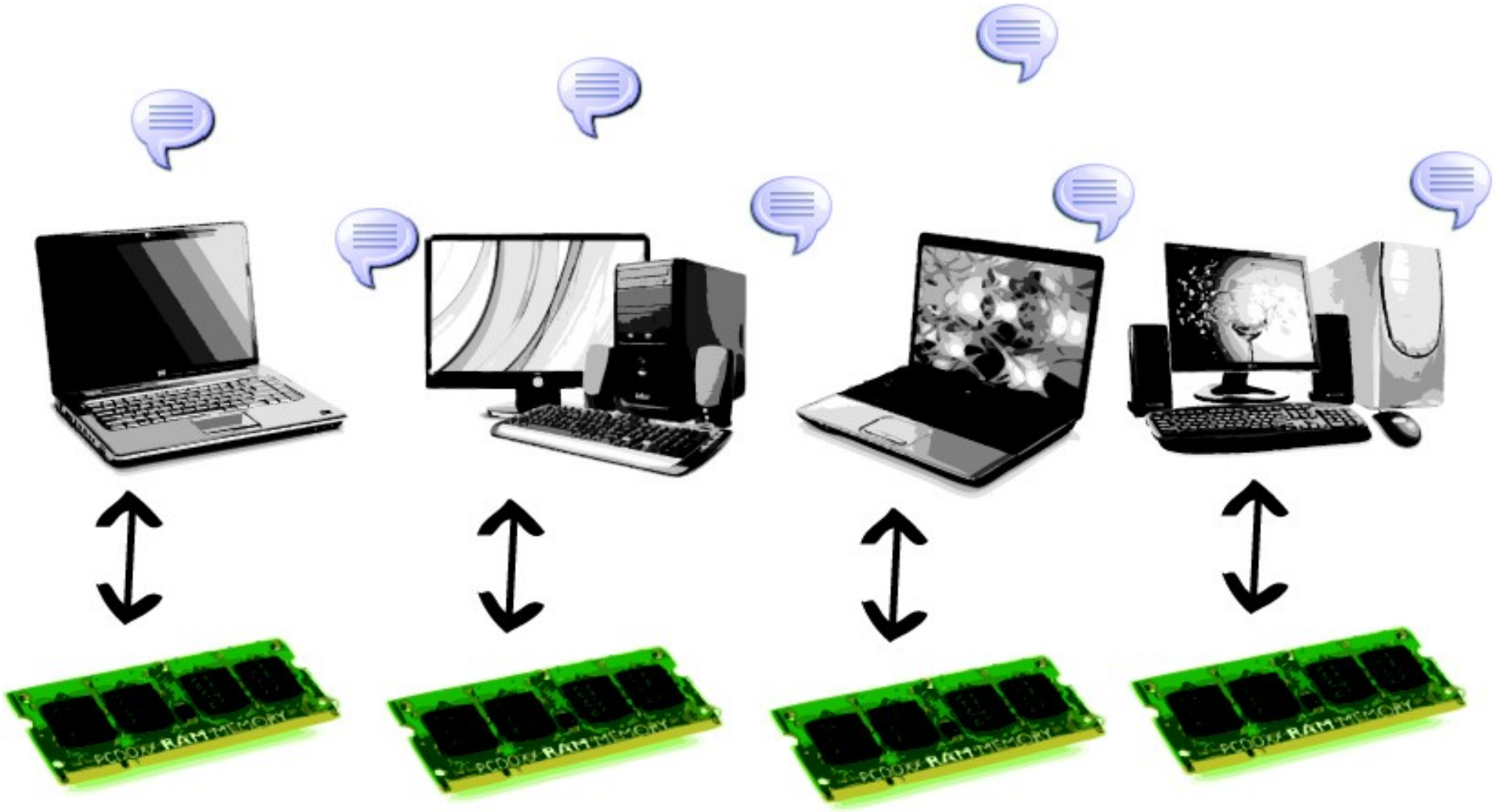
Quando há compartilhamento, **o mesmo recurso está disponível a todas as unidades de execução**. Todas podem (a princípio) acessar os mesmos dados, ao mesmo tempo:





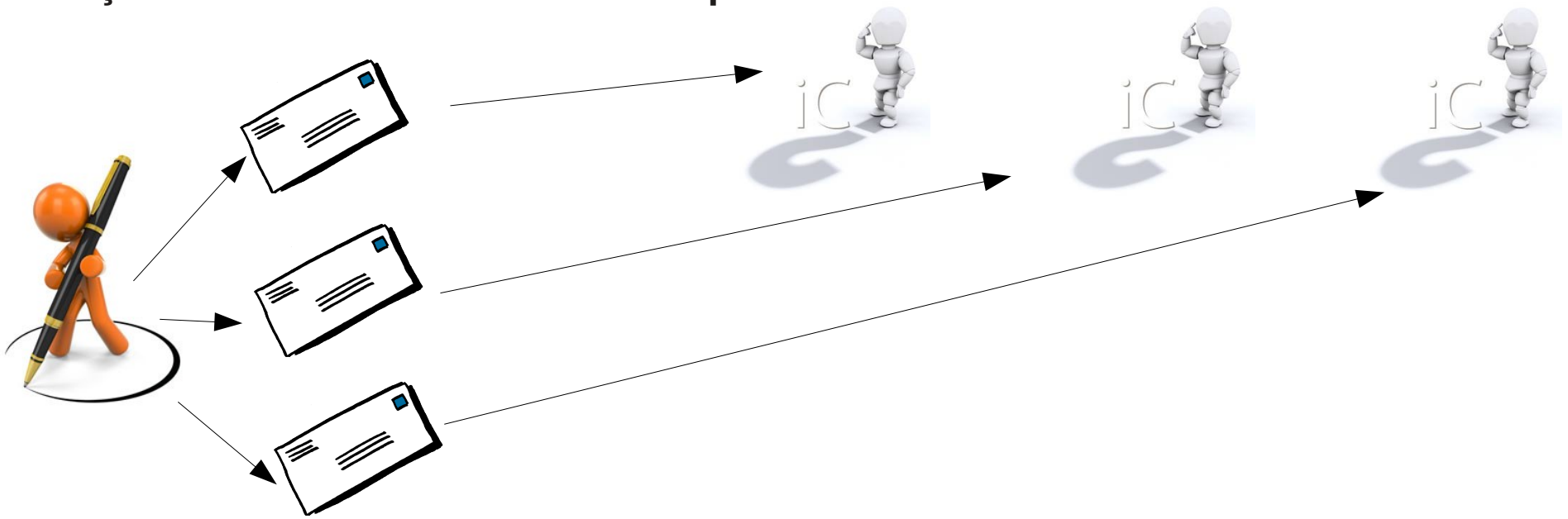
Principais arquiteturas paralelas atuais

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



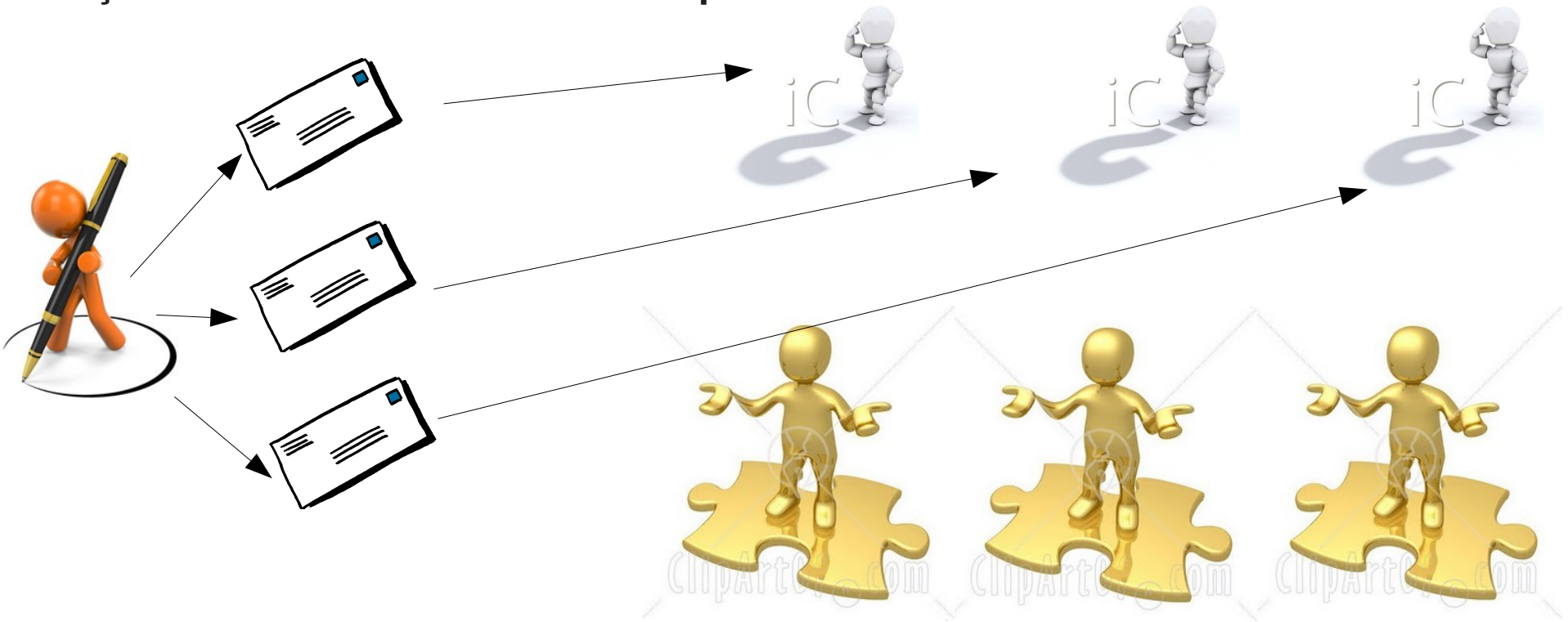
Principais arquiteturas paralelas atuais

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



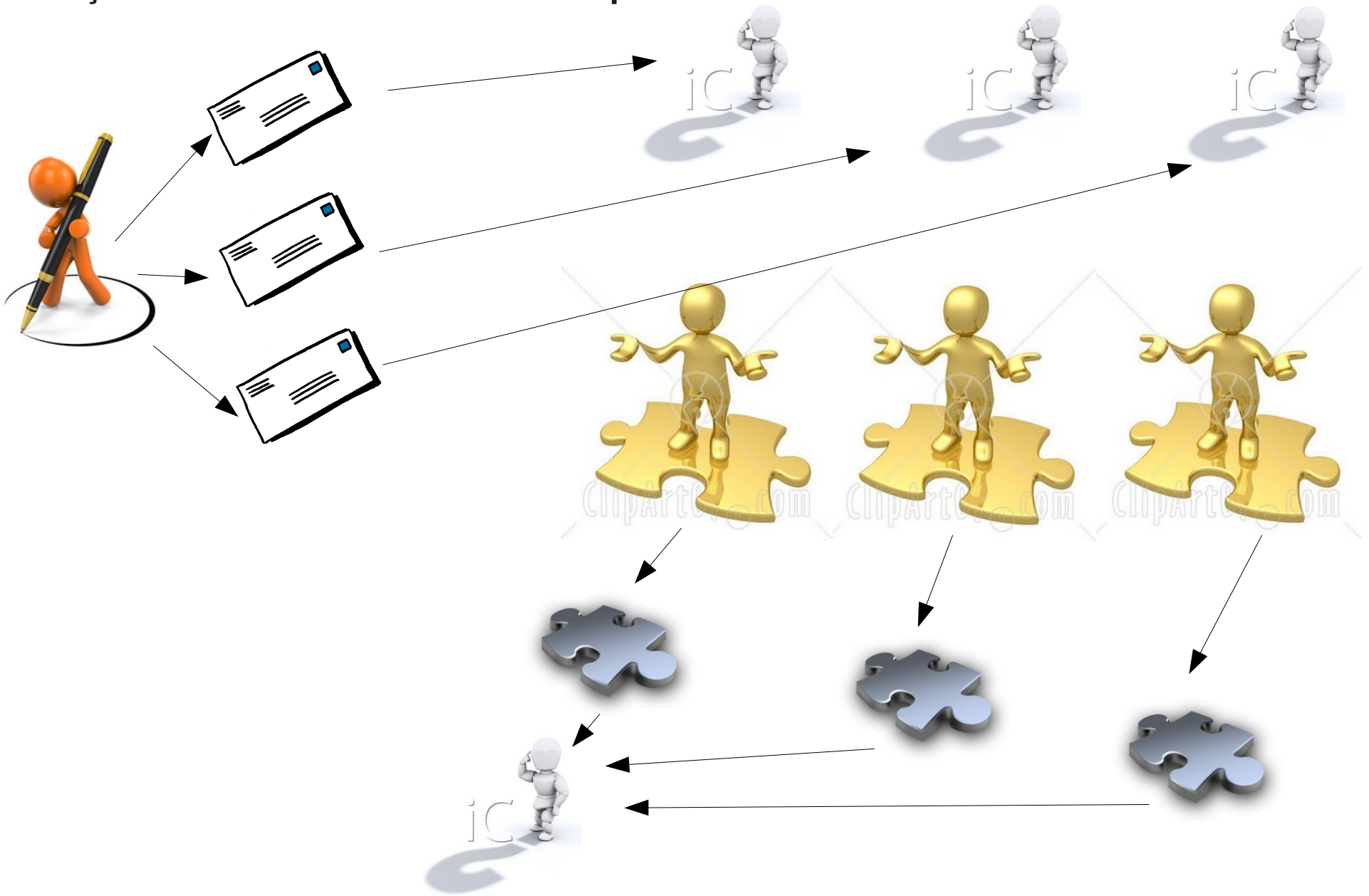
Principais arquiteturas paralelas atuais

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



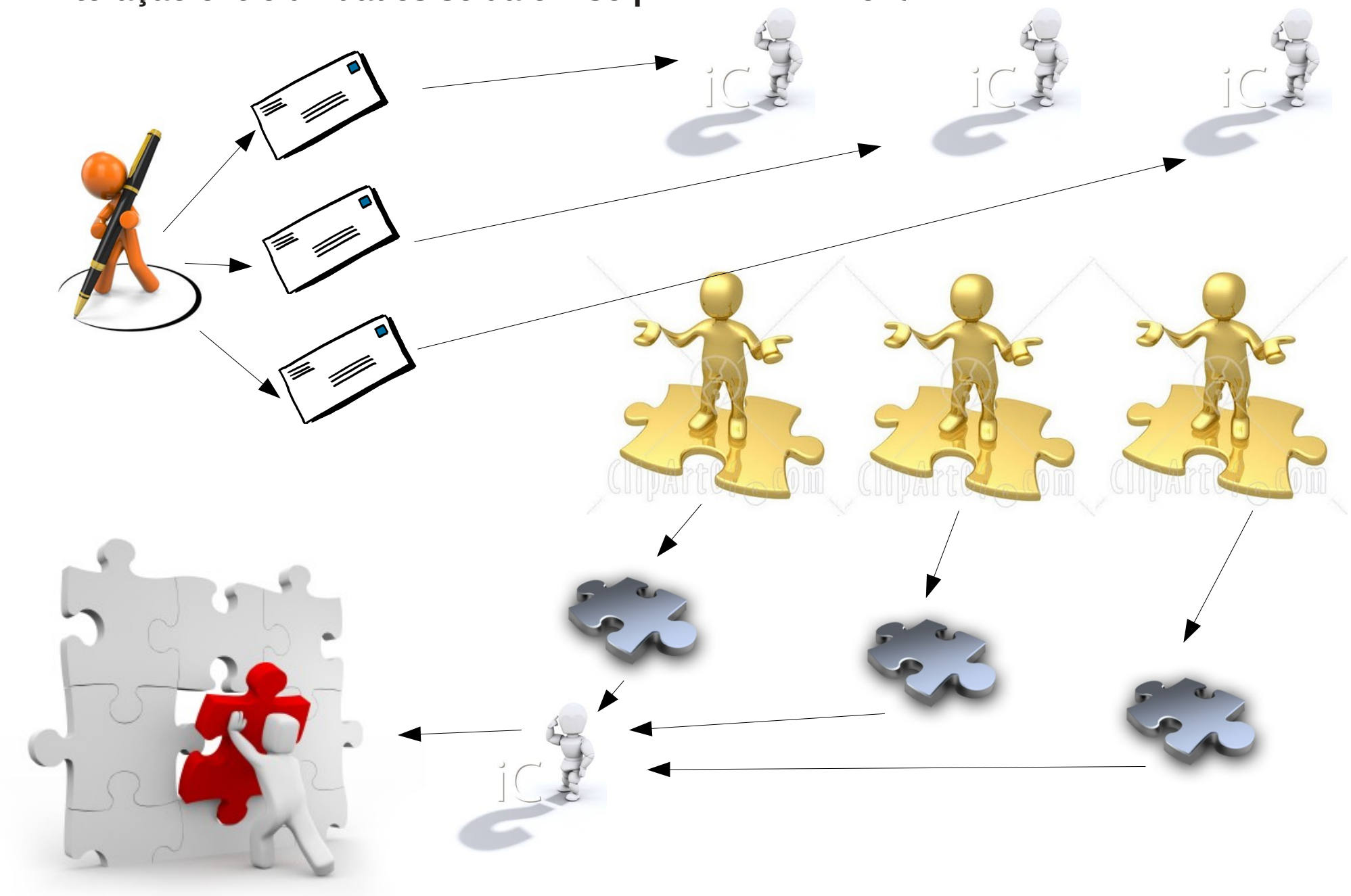
Principais arquiteturas paralelas atuais

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
- OpenMP
- GPUs
- Gerenciamento manual de *threads* (dependente de linguagem e sistema operacional)

Sem memória compartilhada*:

Cada unidade de execução é um **processo**.

- MPI
- Execução de múltiplas instâncias de um programa
- Gerenciamento manual de *processos* (dependente de linguagem e sistema operacional)

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

Comparação de paradigmas

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.
- Muito mais simples que gerenciar processos e sua comunicação diretamente.

Comparação de paradigmas

GPUs

- Usam memória compartilhada.
- Todos os *threads* executam o mesmo código, em dados diferentes.
- Distribuição do trabalho e gerenciamento de *threads* feitos por *driver* / *hardware*.
- Centenas / milhares de núcleos com memória local rápida.
- CUDA (só em GPUs NVIDIA): Software e hardware mais avançados.
- OpenCL (NVIDIA, ATI, CPUs, FPGAs): Universal, código aberto.
- *Kernels* (códigos executados pelas GPUs) têm limitações em complexidade.
- *Software* e *hardware* têm evoluído rapidamente.
- Escrever e testar *kernels* costuma ser trabalhoso.

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.

Comparação de paradigmas

Paralelização extrínseca

- **Programas seriais inalterados (ou quase).**
- **Paralelização perfeita**: computação acelerada linearmente com o número de núcleos (desde que não haja gargalos: rede, memória, disco, etc.)
- Processos diferentes podem estar em computadores diferentes.
- Não é necessária comunicação entre processos.
- Paralelizar exige apenas organizar a divisão do trabalho e a coleta dos resultados.
- Adequada a muitos problemas em astronomia:
  - Calcular modelos com parâmetros diferentes.
  - Processar diferentes conjuntos de dados.

Paralelização indireta

- O programador usa bibliotecas paralelizadas prontas.
- As bibliotecas podem ter sido implementadas por especialistas (cientistas da computação) para fazer seu trabalho da forma mais eficiente possível.
- Casos comuns:
  - Álgebra linear
  - FFT
  - Processamento de imagens
  - Algoritmos comuns (ordenação, contêiners, etc.)
  - Problemas científicos comuns: N-corpos, visualização, CFD, vizinhos (KNN), etc.

Escolhas

OpenMP

- Vantagens:
  - O padrão melhor estabelecido para paralelização.
  - Padrão ainda ativamente desenvolvido.
  - Independente de linguagem, compilador e plataforma.
  - Permite paralelismo de dados e tarefas.
  - Possui tanto construções de baixo nível como de alto nível.
  - Mantém compatibilidade com código serial.
  - Mais fácil que MPI.
- Desvantagens:
  - Limitado a memória compartilhada.
  - Só implementado em Fortran, C, C++.

Escolhas

MPI

- Vantagens:
  - O padrão melhor estabelecido para memória distribuída.
  - Pode ser usado tanto em sistemas de memória compartilhada como de memória distribuída.
  - O padrão mais comum para *clusters*.
  - Padrão ainda ativamente desenvolvido.
  - Permite paralelismo de dados e tarefas.
  - Implementado em várias linguagens (Fortran, C, C++, Python, Java, IDL, etc.).
- Desvantagens:
  - Mais difícil de usar que OpenMP e vetorização.
  - Exige reescrever (possivelmente reestruturar) o código serial.
  - Padrão ainda menos maduro que OpenMP.
  - Ainda varia bastante entre implementações, principalmente na execução (`mpirun`, `mpiexec`, interação com filas).

Escolhas

Vetorização

- Vantagens:
  - Paralelização, se disponível, feita pelo compilador / interpretador.
  - Torna o programa mais organizado e robusto.
  - Permite identificar trechos que podem ser paralelizados de outras formas, explicitamente.
  - Muito mais fácil que as outras formas.
  - Compiladores podem gerar código vetorial para um só núcleo (MMX, SSE, AVX), ou para vários (em vários *threads*).
- Desvantagens:
  - Em geral limitada a memória compartilhada.
  - Limitada a paralelismo de dados.
  - Só se aplica aos casos mais simples de paralelização.
  - Abrangência muito variável entre linguagens.

Escolhas

GPUs

- Vantagens:
  - *Hardware* barato com milhares de núcleos e memória interna rápida.
  - Otimizadas para repetir o mesmo código em grandes conjuntos de dados.
  - Em rápida evolução atual, diminuindo suas limitações.
  - GPUs estão ganhando características de CPUs, e CPUs estão ganhando características de GPUs
  - Várias bibliotecas estão sendo implementadas em GPUs.
- Desvantagens:
  - Uso de CUDA restrito a *hardware* NVIDIA.
  - Tradicionalmente cada GPU só executa um código de cada vez.
  - Aplicação limitada.
  - Desenvolvimento e *debug* mais difíceis.

Escolhas

Paralelização extrínseca

- Vantagens:
  - Fácil de implementar: apenas organizar a execução.
  - Paralelização perfeita.
- Desvantagens:
  - Não diminui o tempo necessário para obter 1 resultado.

Paralelização indireta (uso de bibliotecas)

- Vantagens:
  - Biblioteca desenvolvida e testada por especialistas, fazendo uso eficiente de OpenMP, MPI, GPUs, vetorização, etc.
  - Em geral fácil de usar.
- Desvantagens:
  - Nem todas são gratuitas.
  - Incompatibilidades, falta de variedade.

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:

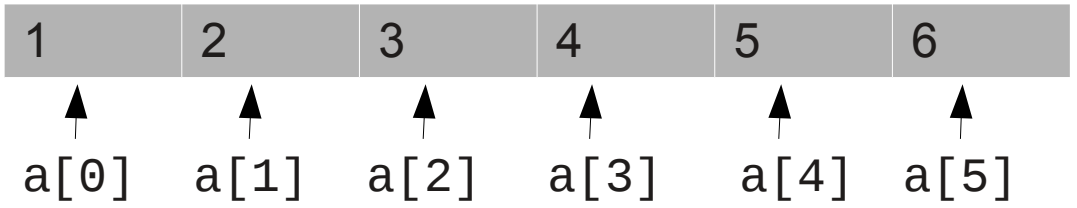
- 1D é muito comum.
- MD é com freqüência desajeitado (2D ocasionalmente não é tão ruim): **IDL e Python+Numpy têm arrays MD de muito alto nível** (adiante).

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.

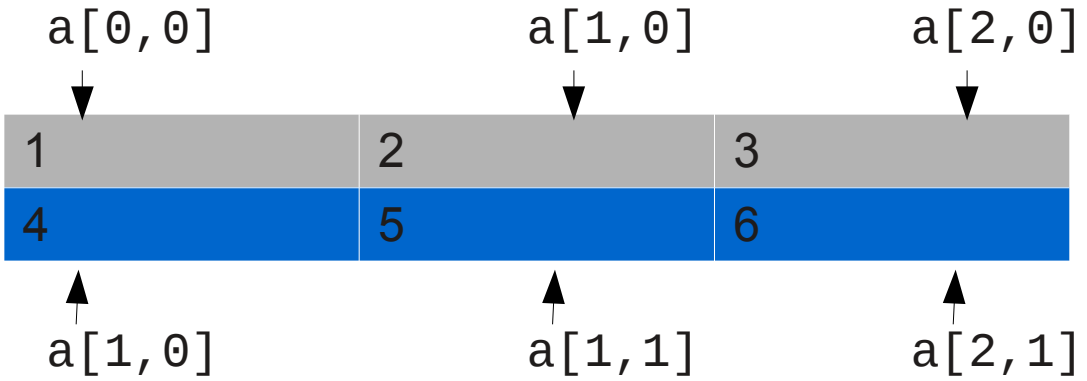
```
IDL> help,a
A               INT       = Array[6]

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)
```

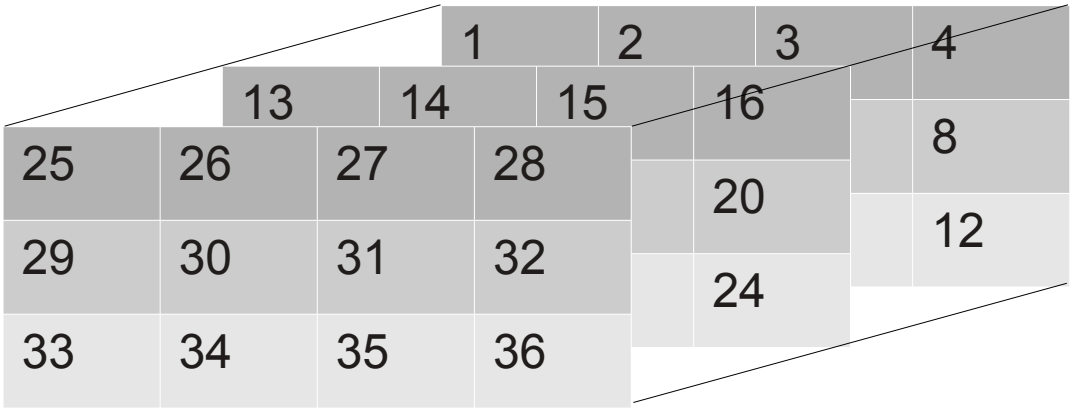
Gera um *array* de tipo ***byte***, com 36 elementos, em 4 colunas, 3 linhas, 3 “páginas”, com valores 0 a 35.

```
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?

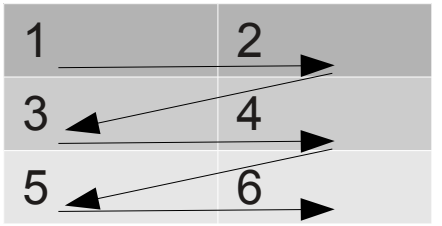
As várias dimensões são varridas ordenadamente. Ex (2D): a[2,3] - 6 elementos:

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
```



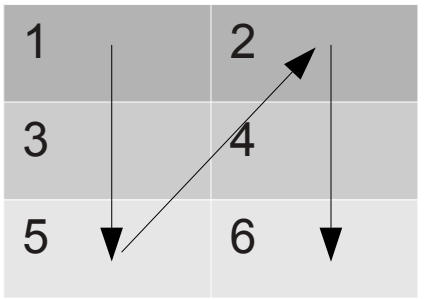
ou

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.

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.

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*.

*Numpy não é (ainda) parte das bibliotecas padrão de Python.

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
  - *Thread pool* (IDL)
  - Em geral habilitado por opções (`-parallel`, `-ftree-parallelize-loops`) em compiladores.

Paralelização extrínseca

Em um único computador, pode ser implementada apenas executando vários conjuntos de programas que recebem argumentos diferentes:

```
[user@computer pp_ex1]$ ls
obs00.fits  obs03.fits  obs06.fits  obs09.fits  obs12.fits  obs15.fits
obs01.fits  obs04.fits  obs07.fits  obs10.fits  obs13.fits  obs16.fits
obs02.fits  obs05.fits  obs08.fits  obs11.fits  obs14.fits  obs17.fits
...
[user@computer pp_ex1]$ run_my_pipeline obs0*.fits &
[user@computer pp_ex1]$ run_my_pipeline obs1*.fits &
[user@computer pp_ex1]$ run_my_pipeline obs2*.fits &
```

Paralelização extrínseca

Em um único computador, pode ser implementada apenas executando vários conjuntos de programas que recebem argumentos diferentes:

```
[user@computer pp_ex1]$ ls
obs00.fits  obs03.fits  obs06.fits  obs09.fits  obs12.fits  obs15.fits
obs01.fits  obs04.fits  obs07.fits  obs10.fits  obs13.fits  obs16.fits
obs02.fits  obs05.fits  obs08.fits  obs11.fits  obs14.fits  obs17.fits
...
[user@computer pp_ex1]$ run_my_pipeline obs0*.fits &
[user@computer pp_ex1]$ run_my_pipeline obs1*.fits &
[user@computer pp_ex1]$ run_my_pipeline obs2*.fits &
```

Em *clusters*, se costuma usar um sistema de filas, o que leva a duas opções:

1) Submeter manualmente um trabalho para cada processo (bash, PBS):

```
[user@computer pp_ex1]$ qsub ./run_my_pipeline obs0*.fits
[user@computer pp_ex1]$ qsub ./run_my_pipeline obs1*.fits
[user@computer pp_ex1]$ qsub ./run_my_pipeline obs2*.fits
```

Paralelização extrínseca

2) Submeter um só trabalho que inicia todos os processos (bash, PBS):

```
[user@computer pp_ex1]$ qsub ./pp_para_ex4.sh
```

pp_para_ex4.sh:

```
#PBS -l walltime=2:00
#PBS -l nodes=3
#PBS -t 1-3

filelist="filelist_$PBS_ARRAYID"
echo "I am process $PBS_ARRAYID of $PBS_NP"
echo "I will process $filelist"
./run_my_pipeline $filelist
```

Resultado:

```
I am process 1 of 3
I will process filelist_1
I am process 2 of 3
I will process filelist_2
I am process 3 of 3
I will process filelist_3
```

Paralelização extrínseca

2) Submeter um só trabalho que inicia todos os processos (GDL, PBS):

```
[user@computer pp_ex1]$ qsub ./pp_para_ex3.sh
```

pp_para_ex3.sh:

```
#PBS -l walltime=2:00
#PBS -l nodes=3
#PBS -t 1-3

gdl -e 'pp_para_ex3'
```

pp_para_ex3.pro:

```
pro pp_para_ex3
;find the files to process
files=file_search('./obs*.fits',count=nfiles)
print,'Found ',nfiles,' files'
;find out how many processes exist and my process number
ntasks=long(getenv('PBS_NP'))
if ntasks eq 0 then ntasks=1
procnum=long(getenv('PBS_ARRAYID'))
for ifile=procnum-1L,nfiles-1,ntasks do begin
  print,'I am process ',procnum,' of ',ntasks
  print,'Processing file ',ifile+1,' of ',nfiles,' : ',files[ifile]
  ;do useful stuff with the file
endfor
end
```

Paralelização extrínseca

2) Submeter um só trabalho que inicia todos os processos (GDL, PBS):

Resultado:

```
Found             7 files
I am process          1 of          3
Processing file       1 of          7 : ./obs_1.fits
I am process          1 of          3
Processing file       4 of          7 : ./obs_4.fits
I am process          1 of          3
Processing file       7 of          7 : ./obs_7.fits
Found             7 files
I am process          2 of          3
Processing file       2 of          7 : ./obs_2.fits
I am process          2 of          3
Processing file       5 of          7 : ./obs_5.fits
Found             7 files
I am process          3 of          3
Processing file       3 of          7 : ./obs_3.fits
I am process          3 of          3
Processing file       6 of          7 : ./obs_6.fits
```

Paralelização extrínseca

2) Submeter um só trabalho que inicia todos os processos (IDL, SLURM):

```
[user@computer pp_ex1]$ qbatch ./pp_para_ex2.sh
```

pp_para_ex2.sh:

```
#!/bin/bash
#SBATCH --job-name=pp_slurmjob
#SBATCH --ntasks=3
#SBATCH --time=10:00
#SBATCH --cpus-per-task=1

srun idl -e 'pp_para_ex2'
```

pp_para_ex2.pro:

```
pro pp_para_ex2
;find the files to process
files=file_search('./obs*.fits',count=nfiles)
;find out how many processes exist and my process number
ntasks=long(getenv('SLURM_NTASKS'))
if ntasks eq 0 then ntasks=1
procnum=long(getenv('SLURM_PROCID'))
for ifile=0L+procnum,nfiles-1,ntasks do begin
  print,'I am process ',procnum+1,' of ',ntasks
  print,'Processing file ',ifile+1,' of ',nfiles,' : ',files[ifile]
  ;do useful stuff with the file
endfor
end
```

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

Algumas referências

Gerais (incluindo OpenMP, MPI e outros)

- *Parallel Programming: for Multicore and Cluster Systems* (2010)
  Rauber e Rünger
  http://www.amazon.com/Parallel-Programming-Multicore-Cluster-Systems/dp/364204817X/

- *An Introduction to Parallel Programming* (2011)
  Peter Pacheco
  http://www.amazon.com/Introduction-Parallel-Programming-Peter-Pacheco/dp/0123742609/

- *An Introduction to Parallel Programming with OpenMP, PThreads and MPI* (2011)
  Robert Cook
  http://www.amazon.com/Introduction-Parallel-Programming-PThreads-ebook/dp/B004I6D6BM/

- *Introduction to High Performance Computing for Scientists and Engineers* (2010)
  Hager e Wellein
  http://www.amazon.com/Introduction-Performance-Computing-Scientists-Computational/dp/143981192X/

- Paralelização - Introdução a vetorização, OpenMP e MPI (2011)
  Paulo Penteado
  Slides de um curso de 4 aulas, código-fonte e um artigo em
  http://www.ppenteado.net/ast/pp_para/

Algumas referências

Cloud / computação distribuída

- *The Application of Cloud Computing to Astronomy: A Study of Cost and Perfomance* (2010)
  Berriman et al.
  http://dx.doi.org/10.1109/eScienceW.2010.10
  E outras referências de um dos autores em
  http://www.isi.edu/~gideon/

- *Using Java for distributed computing in the Gaia satellite data processing* (2011)
  O' Mullane et al.
  http://dx.doi.org/10.1007/s10686-011-9241-6

- *Amazon Elastic Compute Cloud (Amazon EC2)*
  http://aws.amazon.com/ec2