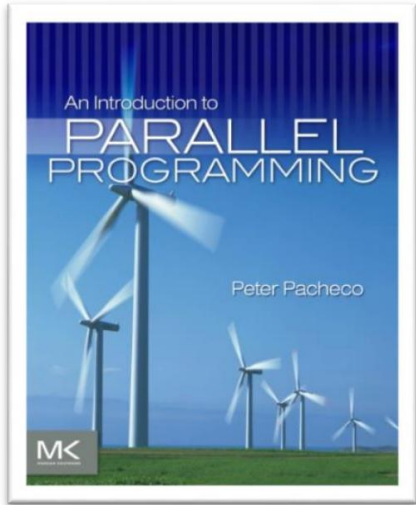


# Uma introdução à programação paralela

## Pedro Pacheco



## Capítulo 1

Por que computação paralela?

Roteiro

184/184

## Tempos de mudança

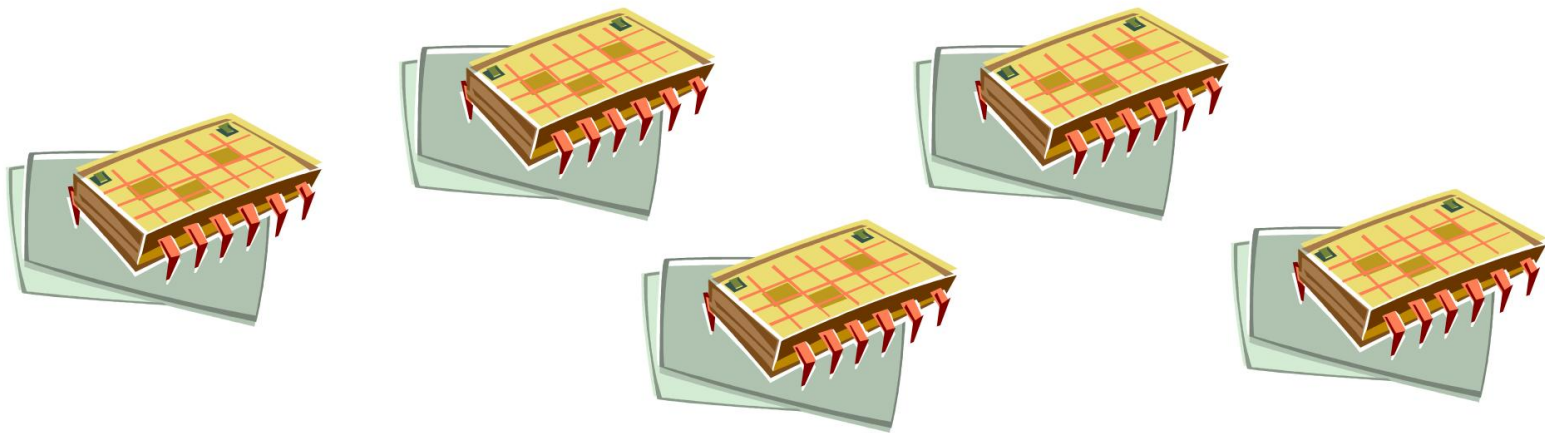
• De 1986 a 2002, os microprocessadores aceleraram como um foguete, aumentando seu desempenho em média 50% ao ano.

• Desde então, caiu para cerca de 20% de aumento ao ano.



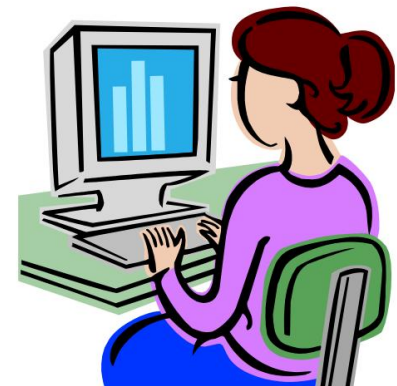
# Uma solução inteligente

• Em vez de projetar e construir mais rápido microprocessadores, coloque vários processadores em um único circuito integrado.



## Agora cabe aos programadores

- Adicionar mais processadores não ajuda muito se os programadores não estiverem cientes disso...
- ... ou não sabe como usá-los.
- Os programas seriais não se beneficiam desta abordagem (na maioria dos casos).



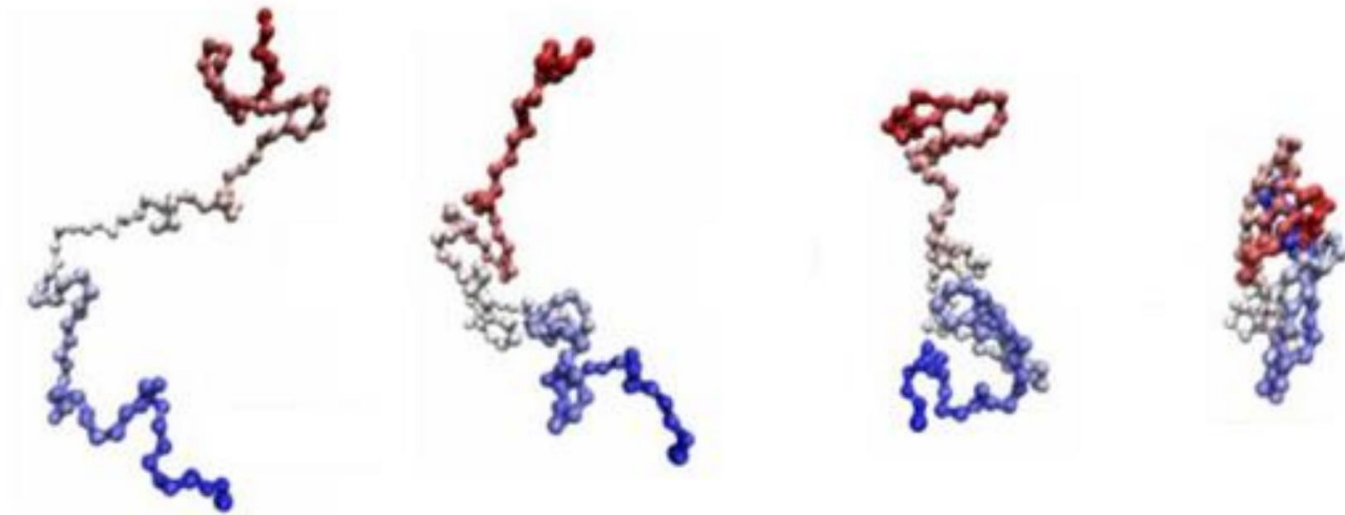
# Por que precisamos de desempenho cada vez maior

- O poder computacional está aumentando, mas também estão aumentando nossos problemas e necessidades computacionais.
- Problemas com os quais nunca sonhamos foram resolvidos devido a avanços passados, como a decodificação do genoma humano.
- Problemas mais complexos ainda aguardam solução.

# Modelagem climática



## Dobramento de proteínas





## Descoberta de drogas



## Pesquisa energética



# Análise de dados



+2.688
+5.000
+1.500
+1.125
+1.062



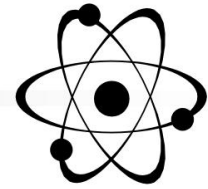
# Por que estamos construindo sistemas paralelos

• Até agora, os aumentos de desempenho foram atribuídos ao aumento da densidade dos transistores.

• Mas há problemas inerentes .



# Uma pequena aula de física

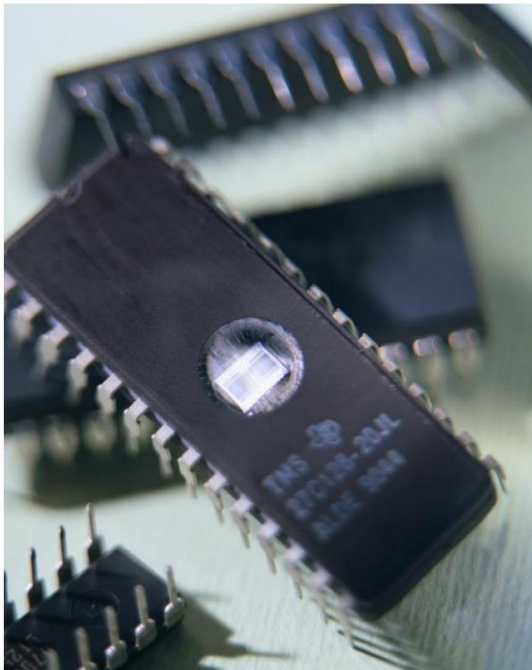


- Transistores menores = processadores mais rápidos.
- Processadores mais rápidos = maior consumo de energia.
- Aumento do consumo de energia = aumento aquecer.
- Aumento de calor = processadores não confiáveis.

# Solução

• Afaste-se dos sistemas de núcleo único para processadores multicore.

• “núcleo” = unidade central de processamento (CPU)



• Apresentando paralelismo!!!

# Por que precisamos escrever programas paralelos

- Executar múltiplas instâncias de um programa serial muitas vezes não é muito útil.
- Pense em executar múltiplas instâncias do seu jogo favorito.
- O que você realmente quer é para para correr mais rápido.



# Abordagens para o problema serial

- Reescrever programas seriais para que fiquem paralelos.
- Escreva programas de tradução que convertam automaticamente programas seriais em programas paralelos.
  - Isso é muito difícil de fazer.
  - O sucesso foi limitado.



# Mais problemas

- Algumas construções de codificação podem ser reconhecido por um gerador automático de programa e convertido em uma construção paralela.
- No entanto, é provável que o resultado seja um programa muito ineficiente.
- Às vezes, a melhor solução paralela é recuar e desenvolver um algoritmo inteiramente novo.

# Exemplo


• Calcule n valores e some-os. • Solução serial:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

# Exemplo (cont.)

• Temos  $p$  núcleos,  $p$  muito menor que  $n$ . •

Cada núcleo realiza uma soma parcial de aproximadamente valores  $n/p$ .



```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
    my_x = Compute_next_value( . . . );
    my_sum += my_x;
}
```

Cada núcleo usa suas próprias variáveis privadas e executa esse bloco de código independentemente dos outros núcleos.

# Exemplo (cont.)

• Depois que cada núcleo completa a execução do código, é uma variável privada `my_sum` contém a soma dos valores calculados por suas chamadas para `Compute_next_value`.

• Ex., 8 núcleos,  $n = 24$ , então as chamadas para `Compute_next_value` retornam:

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

# Exemplo (cont.)

• Depois que todos os núcleos terminam de calcular sua `my_sum` privada, eles formam uma soma global enviando os resultados para um núcleo “mestre” designado que adiciona o resultado final.

# Exemplo (cont.)

```
if (I'm the master core) {  
    sum = my_x;  
    for each core other than myself {  
        receive value from core;  
        sum += value;  
    }  
} else {  
    send my_x to the master;  
}
```

# Exemplo (cont.)

Essencial	0	1	2	3	4	5	6	7
minha_soma	8	19	7	15	7	13	12	14

Soma global

$$8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95$$

Essencial	0	1	2	3	4	5	6	7
minha_soma	95	19	7	15	7	13	12	14

Mas espere!

Existe uma maneira muito melhor de  
calcular a soma global.





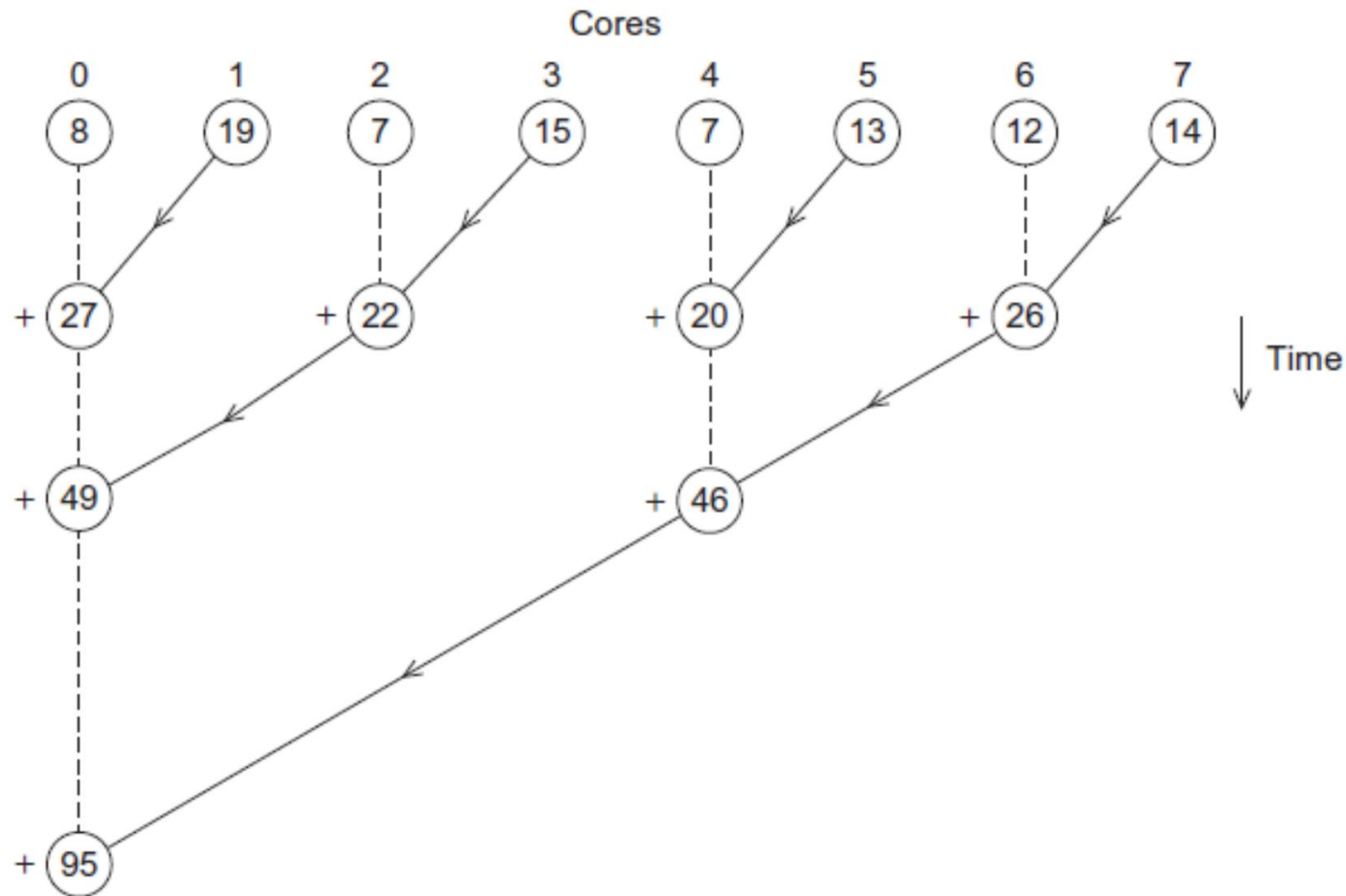
# Melhor algoritmo paralelo

- Não faça com que o núcleo mestre faça todo o trabalho trabalhar.
- Compartilhe-o entre os outros núcleos.
- Emparelhe os núcleos para que o núcleo 0 adicione seu resultado ao resultado do núcleo 1.
- O Core 2 soma seu resultado com o resultado do Core 3, etc.
- Trabalhe com pares de números ímpares e pares de núcleos.

# Melhor algoritmo paralelo (cont.)

- Repita o processo agora apenas com o  
núcleos classificados uniformemente.
  - O núcleo 0 adiciona o resultado do núcleo 2.
  - O Core 4 adiciona o resultado do Core 6, etc.
- 
- Agora os núcleos divisíveis por 4 repetem o  
processo, e assim por diante, até que o núcleo 0 tenha o resultado final.

# Múltiplos núcleos formando um global soma



# Análise

- No primeiro exemplo, o núcleo mestre realiza 7 recepções e 7 adições.
- No segundo exemplo, o núcleo mestre realiza 3 recepções e 3 adições.
- A melhoria é mais do que um fator de 2!

# Análise (cont.)

- A diferença é mais dramática com um maior número de núcleos.
- Se tivermos 1000 núcleos:
  - O primeiro exemplo exigiria que o mestre realizasse 999 recepções e 999 adições.
  - O segundo exemplo exigiria apenas 10 recepções e 10 adições.
- Isso é uma melhoria de quase um fator de 100!

# Como escrevemos programas paralelos?

- Paralelismo de tarefas

- Particionar diversas tarefas realizadas resolvendo o problema entre os núcleos.

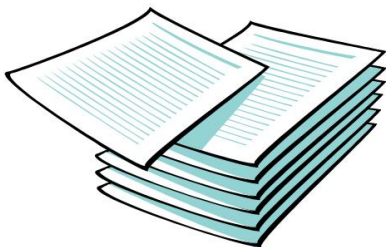
- Paralelismo de dados

- Particione os dados usados na solução do problema entre os núcleos.

- Cada núcleo realiza operações semelhantes em sua parte dos dados.

# Professor P.

15 questões 300  
exames



# Assistentes de avaliação do Professor P

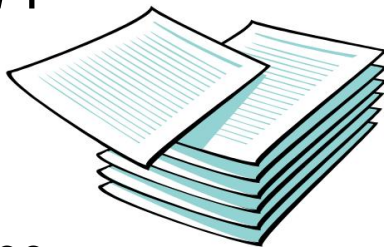




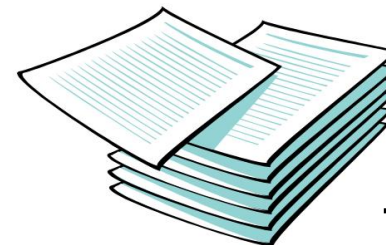
# Divisão de trabalho – paralelismo

de dados

TA#1



100 exames



100 exames

TA#3



100 exames

TA#2

# Divisão de trabalho – paralelismo

de tarefas

TA#1



Perguntas 1 a 5



TA#3

Perguntas 11 a 15



Perguntas 6 a 10

TA#2

## Divisão de trabalho – paralelismo de

dados

```
sum = 0;  
for (i = 0; i < n; i++) {  
    x = Compute_next_value(. . .);  
    sum += x;  
}
```

# Divisão de trabalho – paralelismo de

tarefas

```
if (I'm the master core) {  
    sum = my_x;  
    for each core other than myself {  
        receive value from core;  
        sum += value;  
    }  
} else {  
    send my_x to the master;  
}
```

## Tarefas

- 1) Recebendo
- 2) Adição

## Coordenação

---

- Os núcleos geralmente precisam coordenar seu trabalho.
- Comunicação – um ou mais núcleos enviam suas somas parciais atuais para outro núcleo.
- Balanceamento de carga – compartilhe o trabalho uniformemente entre os núcleos para que um não fique muito carregado.
- Sincronização – como cada núcleo funciona em seu próprio ritmo, certifique-se de que os núcleos não fiquem muito à frente dos demais.

## O que estaremos fazendo

---

• Aprender a escrever programas explicitamente paralelos.

• Usando a linguagem C.

• Usando três extensões diferentes

para C. • Interface de passagem de mensagens (MPI) •

Posix Threads (Pthreads) • OpenMP

## Tipo de sistemas paralelos

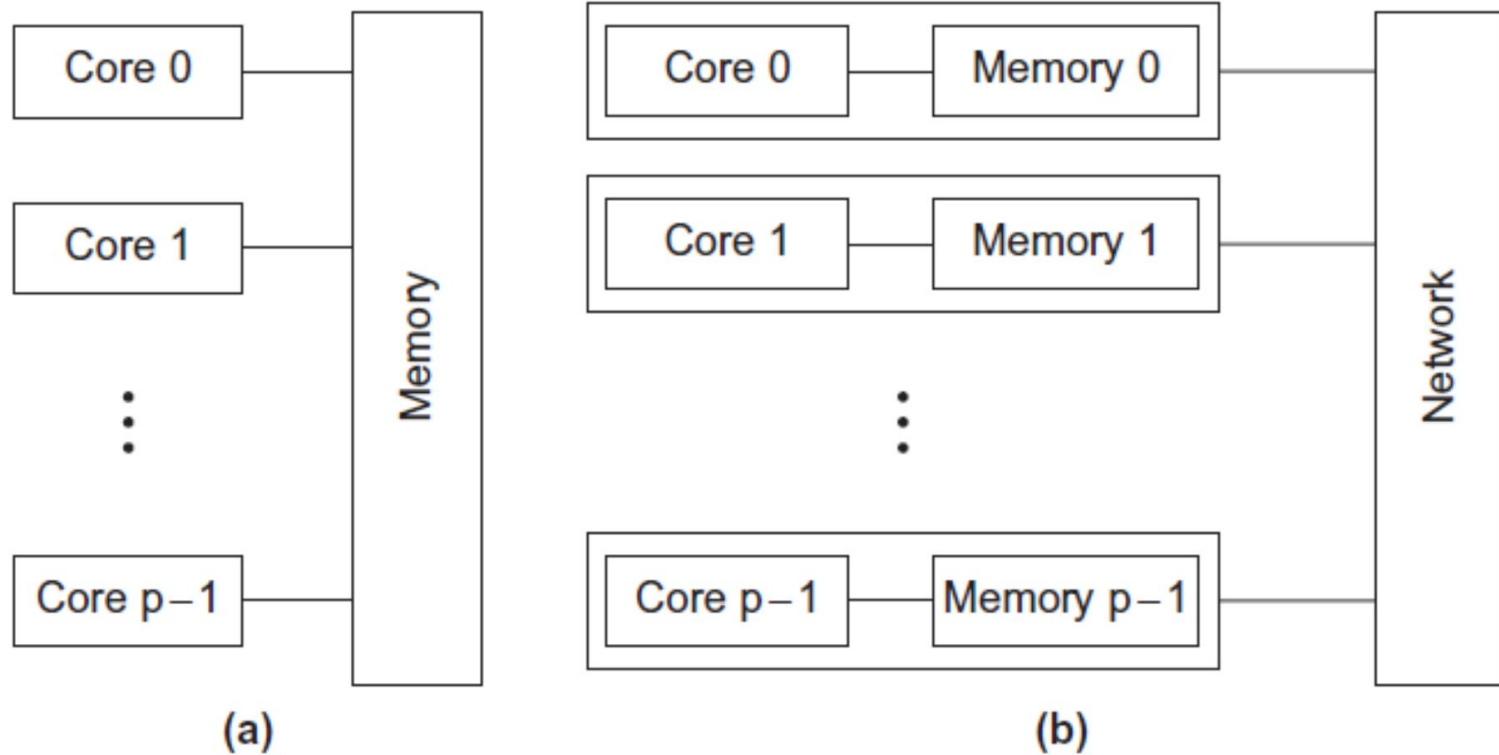
### • Memória compartilhada

- Os núcleos podem compartilhar o acesso ao computador memória.
- Coordene os núcleos fazendo com que eles examinem e atualizem os locais de memória compartilhada.

### • Memória distribuída

- Cada núcleo possui sua própria memória privada.
- Os núcleos devem se comunicar explicitamente enviando mensagens através de uma rede.

# Tipo de sistemas paralelos



Memória compartilhada Memória distribuída



# Terminologia

- Computação simultânea – um programa é aquele em que múltiplas tarefas podem estar em andamento a qualquer momento.
- Computação paralela – um programa é aquele em que múltiplas tarefas cooperam estritamente para resolver um problema
- Computação distribuída – um programa pode necessidade de cooperar com outros programas para resolver um problema.

## Observações Finais (1)

- As leis da física nos levaram às portas da tecnologia multicore.
- Programas seriais normalmente não se beneficiam vários núcleos.
- A geração automática de programas paralelos a partir de código de programa serial não é a abordagem mais eficiente para obter alto desempenho de computadores multicore.

## Observações Finais (2)

- Aprendendo a escrever programas paralelos envolve aprender como coordenar o núcleos.
- Programas paralelos são geralmente muito complexos e, portanto, requerem técnicas e desenvolvimento de programas sólidos.