

MO644/MC970

Introdução

Prof. Guido Araujo

www.ic.unicamp.br/~guido

Adm

- Objetivo
 - Estudar os principais paradigmas de programação paralela e suas
 - OpenMP, Pthreads, CUDA e
 - Introduction to Parallel Programming, Peter Pacheco
- Avaliação
 - Tarefas simples de programação (10)
 - Um projeto de tamanho médio
- Graduação
 - $M = 0.7 * \text{Média (Tarefas)} + 0.3 * \text{Projeto}$
- Pós-graduação
 - $M = 0.6 * \text{Média (Tarefas)} + 0.2 * \text{Projeto} + 0.2 * \text{Exame}$

Questões chave

- Por que precisamos de desempenho maior?
- Por que é preciso construir sistemas paralelos?
- Por que precisamos escrever programas paralelos?
- Como podemos escrever programas paralelos?
- O que nós iremos fazer neste curso?
- Concorrente, paralelo, distribuído!

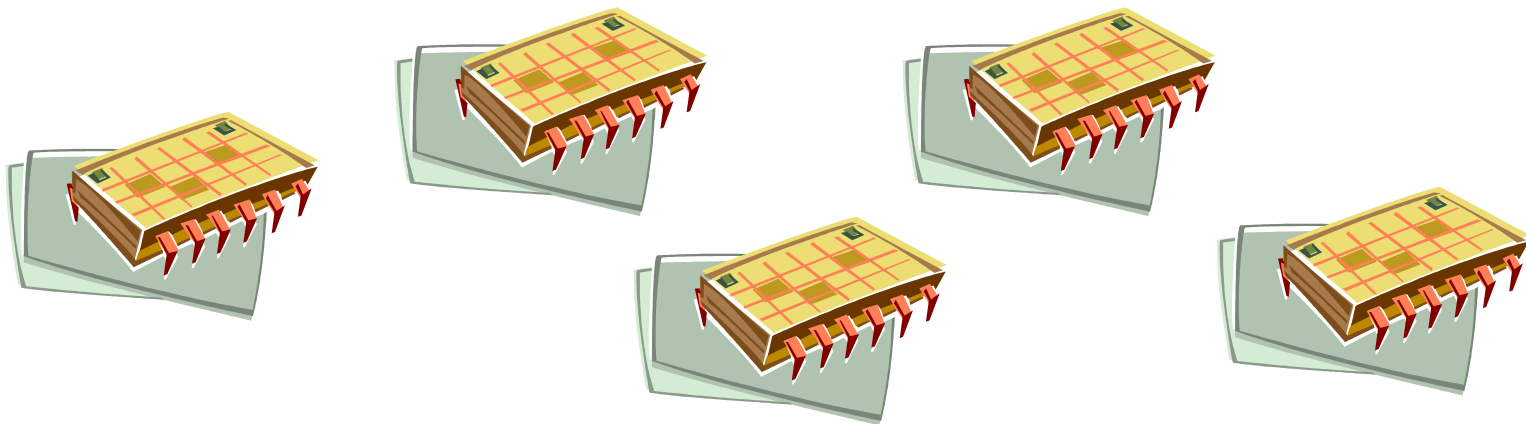
Tempos de mudança

- De 1986 - 2002, microprocessadores aumentaram o seu desempenho como um foguete, uma média de 50% ao ano!
- Desde então, tem caído para cerca de 20% de aumento por ano.



Solução Inteligente

- Em vez de projetar e construir microprocessadores mais rápidos, colocar múltiplos processadores em um único circuito integrado.



Agora é com os programadores...

- Adicionando mais processadores não ajuda muito se os programadores não estão cientes deles ...
- ... Ou não sabe como usá-los.?
- Programas seriais não se beneficiam desta abordagem (na maioria dos casos).



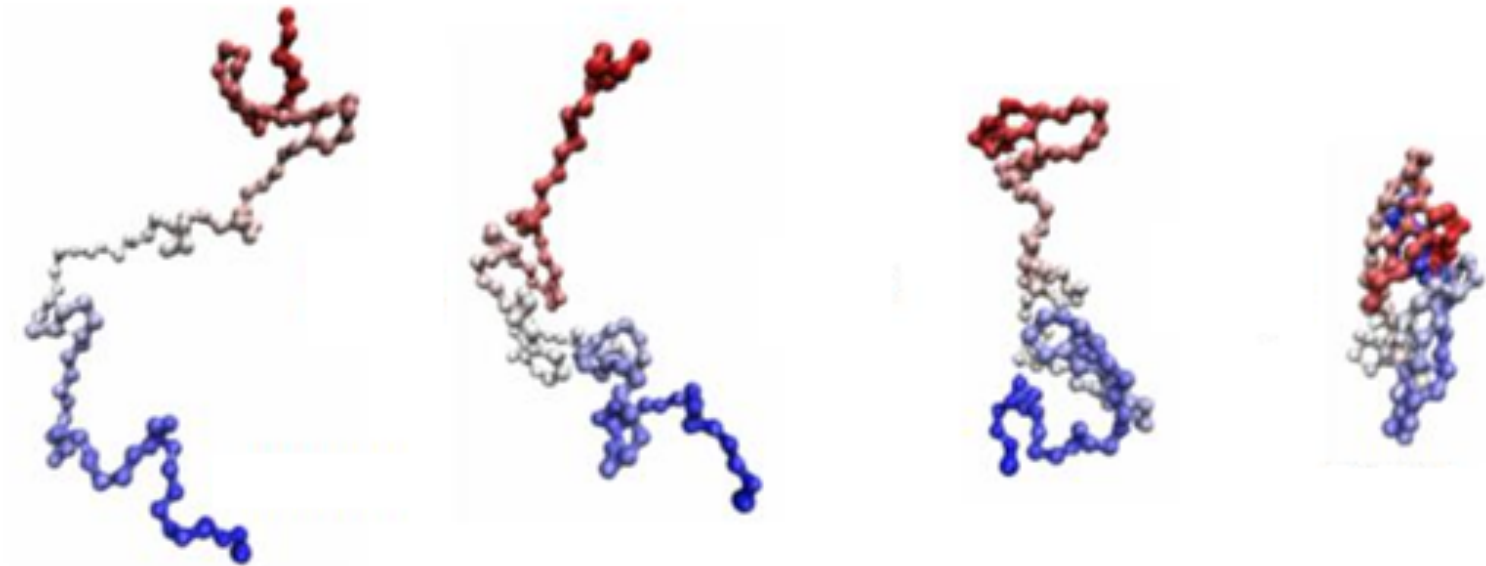
Por quê precisamos de mais desempenho?

- O poder computacional está aumentando, mas também está a complexidade dos problemas.
- Problemas que nunca sonhamos em resolver (ex. decodificação do genoma humano), têm sido resolvidos devido à programação paralela.
- Problemas mais complexos ainda estão esperando para serem resolvidos (ex. Brain Mapping).

Mudança climática



Dobradura de proteínas



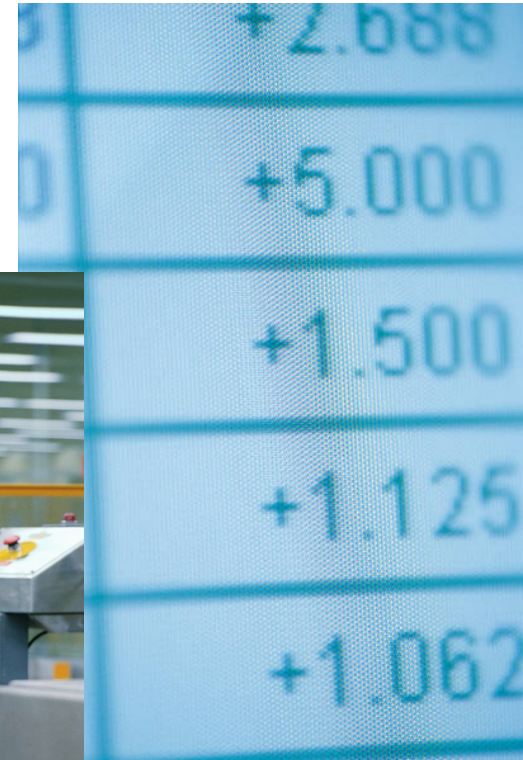
Descoberta de novos remédios



Pesquisa energética



Análise de dados

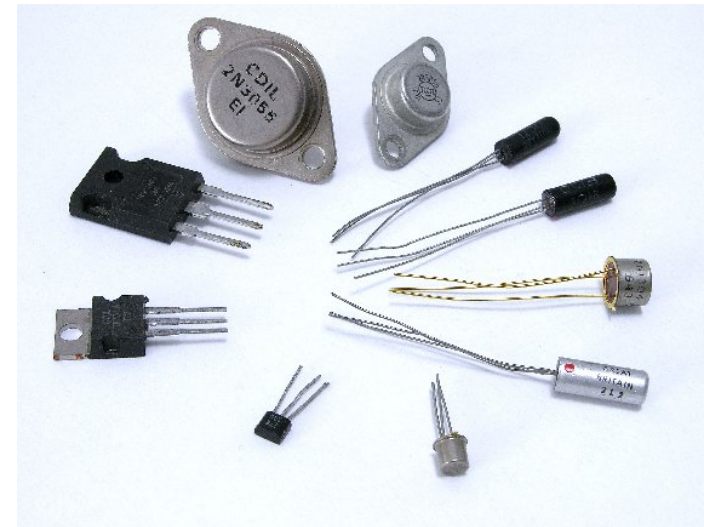


+2.688
+5.000
+1.500
+1.125
+1.062

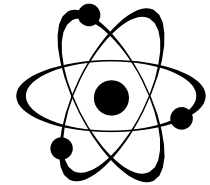
A close-up of a digital display showing a list of positive numerical values. The values are displayed in a grid, with the first row containing +2.688, the second row containing +5.000, the third row containing +1.500, the fourth row containing +1.125, and the fifth row containing +1.062.

Por quê construir sistemas paralelos?

- Até agora, os aumentos de desempenho foram resultado do aumento na densidade de transistores.
- Mas existem problemas...

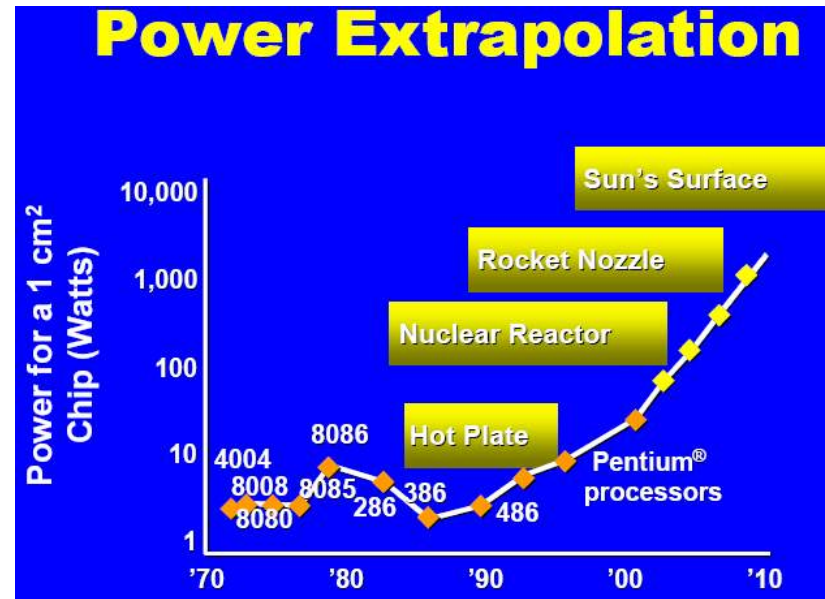
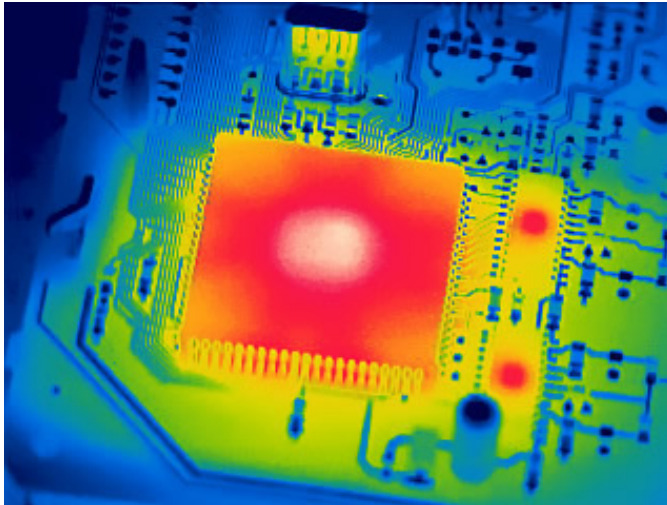


Um pouco de Física



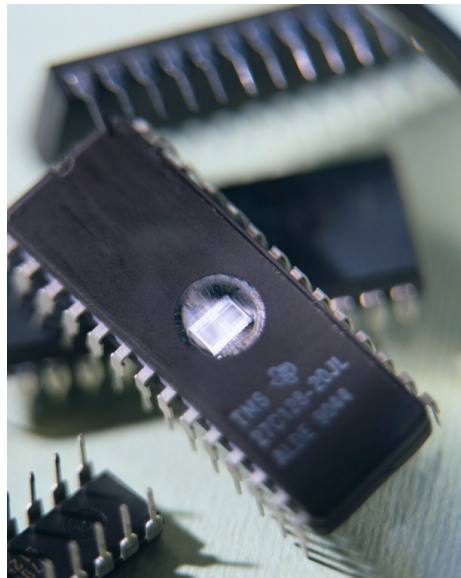
- Transistores menores = processadores mais rápidos.
- Processadores mais rápidos = aumento do consumo de energia.
- Aumento no consumo de energia = aumento no calor.
- Calor aumentado = processadores não-confiáveis.

Cenário em 2005



Solução

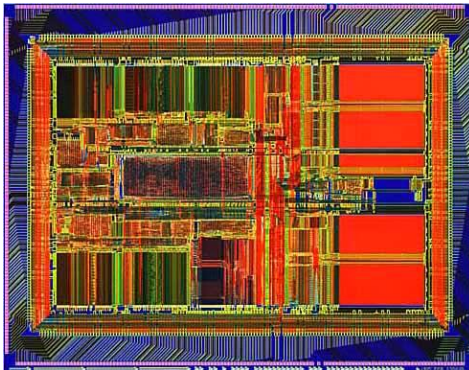
- Procurar alternativas para sistemas single-core
- Processadores multicore.
- "Núcleo" = unidade central de processamento (CPU)



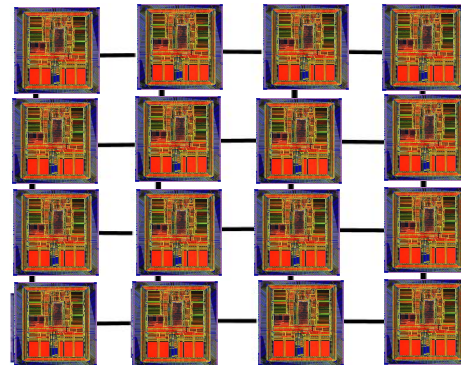
Introduzindo paralelismo!!

Multicore

- Dada uma mesma área de silício
- Um único processador: 4 GHz, alto Watts/cm²
- Vários (16) núcleos: 2 GHz, baixo Watts/cm²

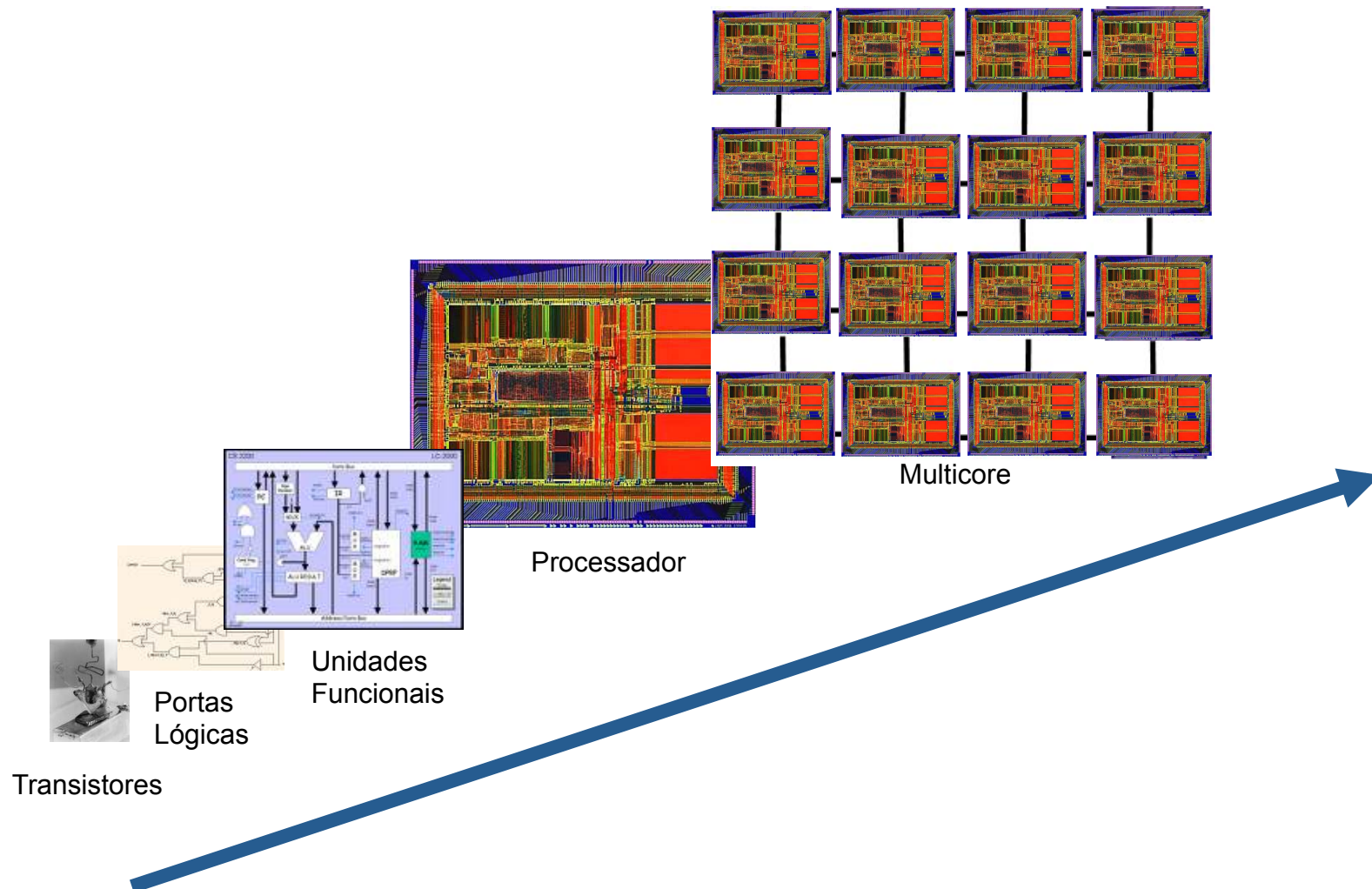


Microrrocessador



Multicore

Evolução natural



Por quê é preciso escrever programas paralelos ?

- Executar várias instâncias de um programa serial, muitas vezes, não é útil.
- Por exemplo, executar várias instâncias do seu jogo favorito.
- O que você realmente quer é que ele execute mais rapidamente



Soluções para um problema serial

- Reescrever programas seriais, de modo que eles se tornem paralelos.?
- Escrever programas de tradução que convertam automaticamente programas seriais em paralelos.
 - Isto é muito difícil de se fazer.
 - O sucesso tem sido limitado.

Mais problemas

- Alguns trechos de um programa podem ser reconhecidos por um gerador automático de programa, e convertidos em um trecho paralelo.
- No entanto, é provável que o resultado seja um programa bem ineficiente.
- Às vezes, a melhor solução paralela é dar um passo atrás e desenvolver um algoritmo inteiramente novo.


Exemplo

- Calcular os valores de n e somá-los
- Solução serial:


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

Exemplo (cont.)

- Temos p núcleos, p muito menor do que n .
- Cada núcleo realiza uma soma parcial de aproximadamente n/p valores.



```
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 este bloco de código independentemente dos outros núcleos.

Exemplo (cont.)

- Depois que cada núcleo conclui a execução do código, uma variável privada `my_sum` contém a soma dos valores calculados por suas chamadas à `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.)

- Quando todos os núcleos terminarem de computar `my_sum`, eles determinam uma soma global enviando os resultados parciais para o núcleo “mestre” que soma o valor 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.)

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Soma global

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

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

Eureca!!

Mas espere aí!

Há uma maneira muito melhor
Para calcular a soma global.



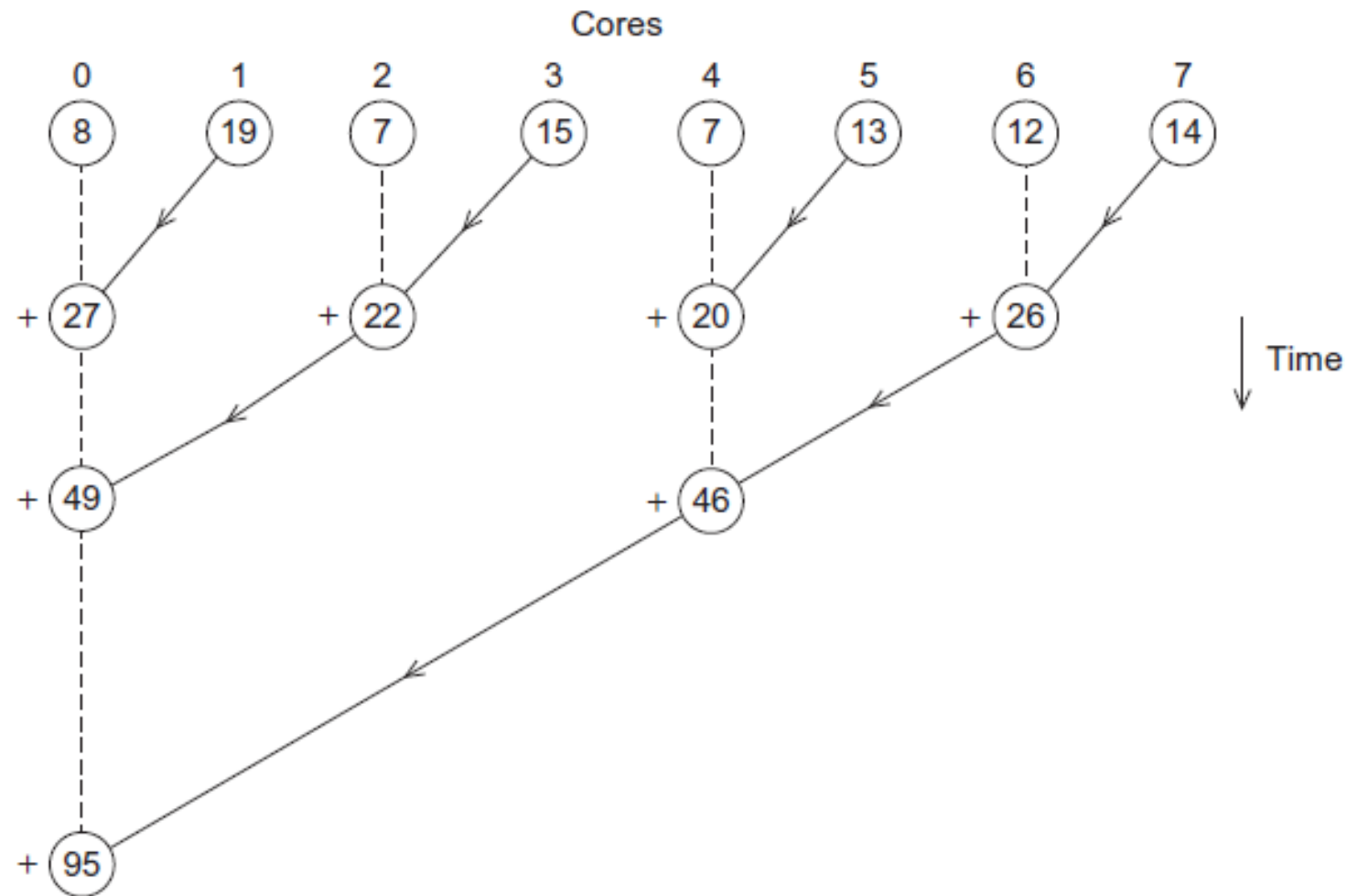
Um algoritmo paralelo melhor

- Não deixar o núcleo mestre (0) fazer todo o trabalho.
- Compartilhá-lo entre os outros núcleos.
- Emparelhar os núcleos de modo que o núcleo 0 acrescenta a sua soma à soma do núcleo 1.
- Núcleo 2 adiciona a sua soma ao resultado do núcleo 3, etc
- Organizar os núcleos em conjuntos par-ímpar.

Um algoritmo paralelo melhor (cont.)

- Repita o processo agora apenas com os núcleos pares.
- Núcleo 0 soma o resultado do núcleo 2.
- Núcleo 4 soma o resultado do núcleo 6, etc
- Repita para os núcleos divisíveis por 4, e assim por diante, até núcleo 0 conter o resultado final.

Usando múltiplos núcleos para calcular uma soma global



Análise

- No primeiro exemplo, o núcleo mestre executa 7 adições e recebe 7 “recepções”.
- No segundo exemplo, o núcleo mestre executa 3 adições e recebe 3 “recepções”.
- O desempenho melhora mais do que 2x!

Análise (cont.)

- A diferença é mais dramática quando se usa um número maior de núcleos.
- Se tivéssemos mil núcleos:
 - O primeiro exemplo exigiria o mestre executasse 999 adições e 999 “recepções”.
 - O segundo exemplo exigiria apenas 10 adições e 10 “recepções”.
- Isso é uma melhoria de quase 100x!

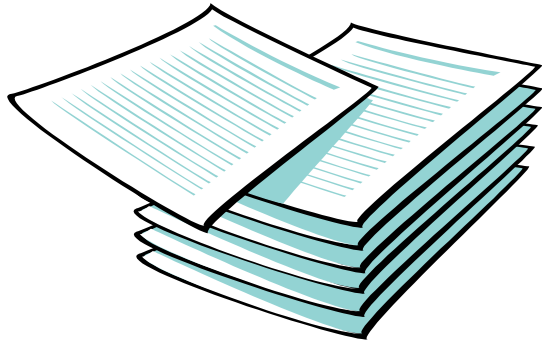
Como escrever programas paralelos?

- *Task parallelism*
 - Divide o problema em várias tarefas que são distribuídas entre os núcleos.
- *Data parallelism*
 - Divide os dados do problema entre os núcleos.
 - Cada núcleo executa tarefas similares na sua parte do dado.

Professor P

15 questões

300 provas



PEDs do Professor P



PED1

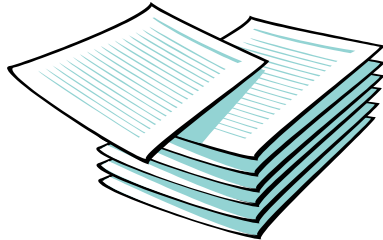
PED2

PEd3

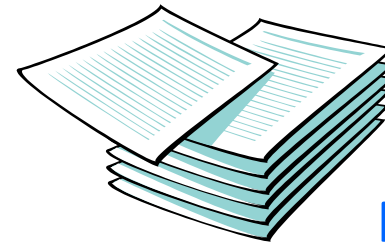
Divisão do trabalho

Paralelismo de Dados

PED1

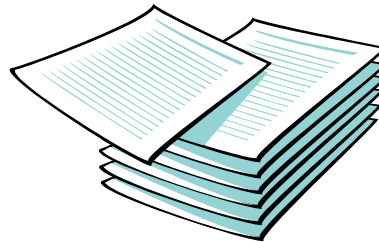


100 exams



PED3

100 exams



PED2

100 exams

Divisão do trabalho

Paralelismo de Tarefas

PED1



Questões 1 - 5



PED3

Questões 11 - 15



PED2

Questões 6 - 10

Divisão do trabalho

Paralelismo de Dados

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

Divisão do 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) Receber
- 2) Somar

Coordenando o trabalho

- Núcleos geralmente precisam coordenar o trabalho.
- **Comunicação** - um ou mais núcleos enviam suas somas parciais para outros núcleos.
- **Balanceamento de carga** - partes do trabalho são distribuídas uniformemente entre os núcleos de modo que um núcleo não fica sobrecarregado.
- **Sincronização** - cada núcleo trabalha em seu próprio ritmo; é preciso garantir que um núcleo não fica muito à frente dos outros.

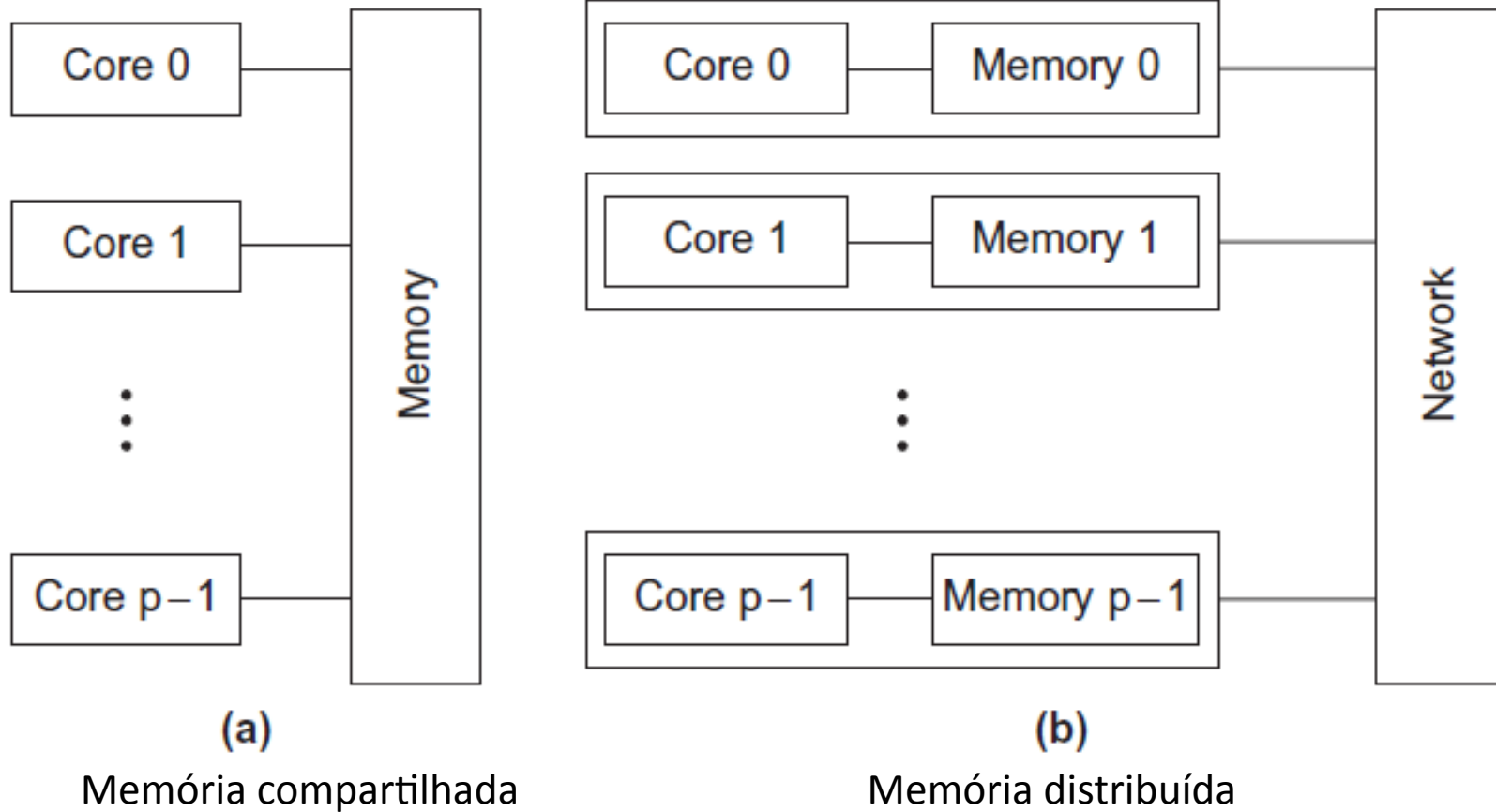
O que nós faremos

- Aprender a escrever programas que são explicitamente paralelo.
- Usando três diferentes extensões para C.
 - Message Passing Interface (MPI)
 - Posix Threads (Pthreads)
 - OpenMP
- Escrever programas paralelos para GPU (CUDA)
- Escrever programa paralelos para nuvem (Hadoop map-reduce)

Tipos de sistemas paralelos

- Memória Compartilhada
 - Os núcleos podem compartilhar o acesso à memória do computador.
 - Coordena os núcleos, permitindo-os examinar e atualizar posições de memória compartilhada.
- Memória Distribuída
 - Cada núcleo tem sua memória, privada.
 - Os núcleos se comunicam explicitamente, enviando mensagens através de uma rede.

Tipos de sistemas paralelos



Terminologia

- **Computação concorrente** – program possui múltiplas tarefas que podem estar em execução a qualquer instante.
- **Computação paralela** – o programa possui várias que tarefas que cooperaram estreitamente para resolver um problema
- **Computação distribuída** – o programa pode precisar cooperar com outros programas para resolver um problema.

Conclusões (1)

- As leis da física nos trouxeram a tecnologia multicore.
- Programas seriais normalmente não se beneficiam de múltiplos núcleos.
- Geração automática de código do programa paralelo a partir de um programa serial não é a abordagem mais eficiente para obter um bom desempenho em computadores multicore.

Conclusões (2)

- Aprender a escrever programas paralelos está relacionado a aprender a coordenar os núcleos.
- Programas paralelos são usualmente complexos e demandam boas técnicas de programação.