Computação Paralela[1]

Nivio Ziviani

[1] Conjunto de transparências elaborado por Nivio Ziviani e João Caram

Computação Paralela

Processo de resolver problemas usando computadores paralelos

Processamento Paralelo

Manipulação concorrente de itens de dados pertencentes a um ou mais processos envolvidos na solução de um único problema.

Computador Paralelo

Computador com múltiplos processadores capaz de realizar processamento paralelo.

Necessidade de Alto Desempenho

- Previsão de Tempo
  - Atmosfera:

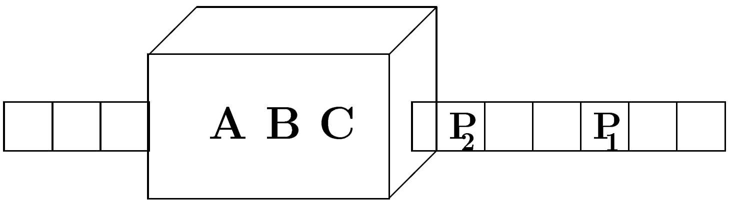
    Altitude, latitude, longitude
  - Grade tridimensional de $450km$ de um lado, por 24 horas:

    100 bilhões de operações

    (Cray 1, 100 Megaflops $\Rightarrow$ 100 minutos)
- Processamento de Imagens
  - 1 figura: 6000×6000 pixels^2 $\times$ 8 bits $=$ 288 MBytes
- Aerodinâmica
  - Troca túnel de vento por simulação

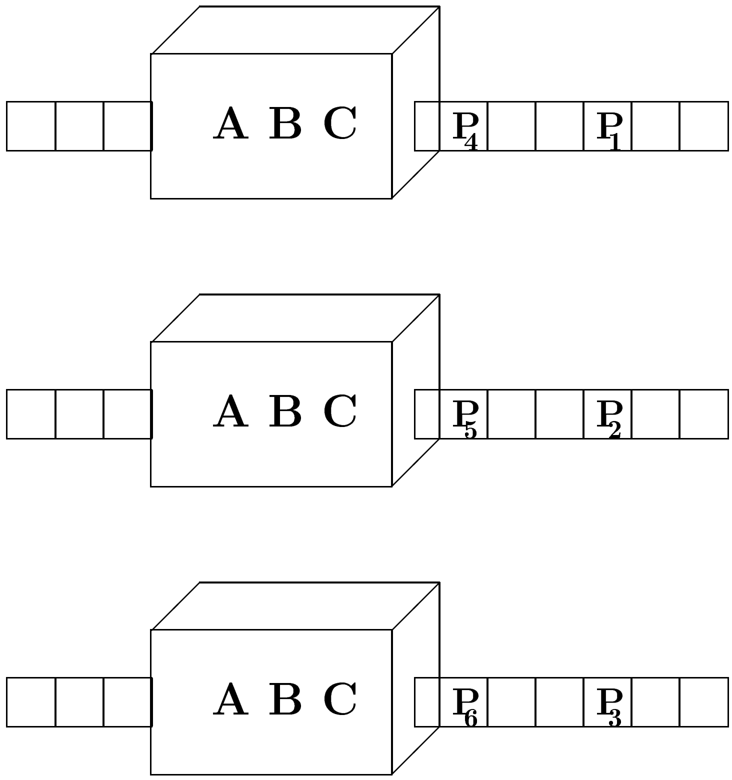
Como aumentar concorrência de uma computação?

- Paralelismo de dados ("data parallelism")
  - Uso de múltiplas unidade para aplicar a mesma operação (simultaneamente) a elementos de conjuntos de dados.
- Paralelismo de controle ("control parallelism") ou "Pipelining"
  - Computação é dividida em estágios ou segmentos
  - Saída de um segmento é entrada para o próximo segmento



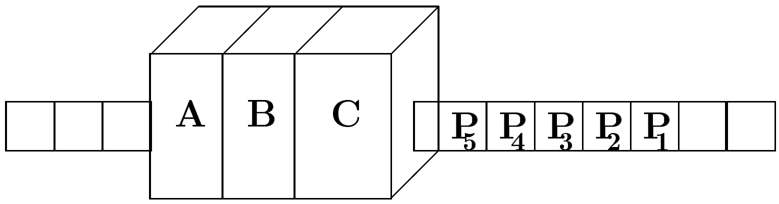
Trabalho em 3 etapas: A, B, C

Paralelismo de Dados



3 produtos a cada 3 unidades de tempo

Paralelismo de Controle



"Pipeline" de 3 segmentos

- 3 máquinas separadas
- Cada máquina cuida de uma subtarefa
- Primeiro produto em 3 unidades de tempo
- Produtos seguintes em unidades de tempo subsequentes

Problemas Reais

- Paralelismo de dados
- Paralelismo de controle
- Problema de relações de precedência

Exemplo: Jardinagem Rápida LTDA

1. Cortar grama
2. Aparar laterais gramado
3. Limpar jardim
4. Verificar irrigadores

- Paralelismo controle: 1, 2, 3 simultaneamente
- Paralelismo dados: mais de uma pessoa em cada <1,2,3>
- Teste irrigadores: somente após o término de 1, 2, 3

Crivo de Eratóstenes

Algoritmo clássico para obter números primos $\leq n$

2 3 4 5 6 7 8 9 10

- Remove números compostos através dos múltiplos dos primos 2, 3, 5, ...
- Termina quando múltiplos do maior primo $\leq \sqrt{n}$ é obtido

Implementação Sequencial

- Arranjo de booleanos
- Inteiro para primo corrente
- Inteiro para controle do anel

Porquê Crivo de Eratóstenes não é útil para testar se um número é primo?

Crivo de Eratóstenes

Implementação Paralela - Paralelismo de Controle

Memória compartilhada:
- Arranjo de booleanos
- Primo corrente

Cada Processador:
- Inteiro para controlar "loop"

Algoritmo:

Todo processador procura próximo primo e marca seus múltiplos

Problemas:
1. Dois processadores podem usar o mesmo primo para percorrer crivo
   - Perda de tempo (não causa erro)
2. Um processador pode marcar múltiplos de um número composto

Crivo de Eratóstenes

Implementação Paralela - Paralelismo de Dados

Algoritmo:
Mesma operação em todos os processadores

- Todos os primos $\leq \sqrt{n}$ no primeiro processador
- Primeiro processador encontra próximo primo e envia para todos os outros

Extensibilidade (“Scalability”)

Algoritmo Extensível

Nível de paralelismo cresce linearmente (pelo menos) com o tamanho do problema

Arquitetura Extensível

Mesmo desempenho por processador na medida que o número de processadores cresce

Importância:

Permitem resolver problemas maiores com o mesmo tempo, comprando mais processadores

Paralelismo de dados mais extensível que paralelismo de controle:

- Paralelismo de controle: constante, independe do tamanho do problema
- Paralelismo de dados: função crescente do tamanho do problema

Taxonomia de Flynn para Arquiteturas

SISD: Single Instruction Single Data
(Computadores Seriais)

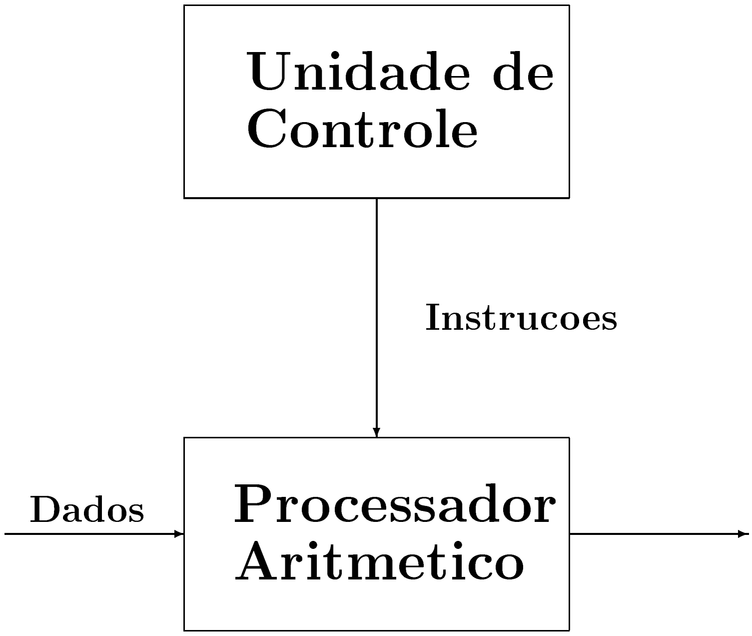
SIMD: Single Instruction Multiple Data
(Varios processadores sincronizados)

MISD: Multiple Instruction Single Data
(Não existem computadores)

MIMD: Multiple Instruction Multiple Data
(Multiprocessamento com interação entre UCPs)

SISD - Single Instruction Single Data

Computador de Von Neuman



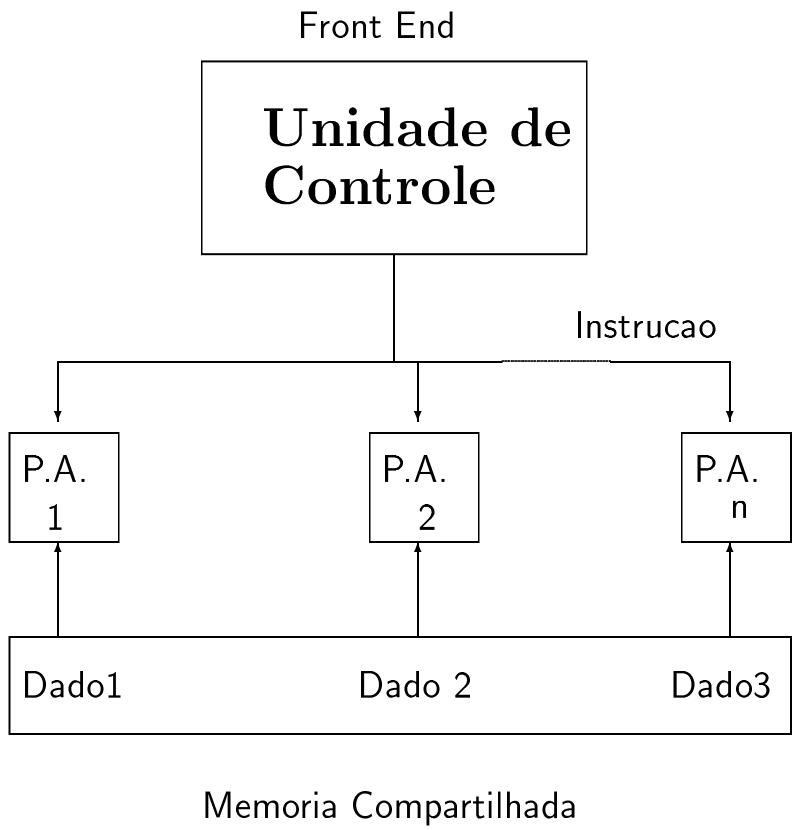
SIMD - Single Instruction Multiple Data

Algoritmos SIMD assumem memória global compartilhada, com acesso por qualquer processador a custo constante.

Componentes:

- Unidade de controle
  - Programa (executa parte serial; emite instruções paralelas)
- Memória Global Compartilhada
- Conjunto de RAM's

SIMD - Single Instruction Multiple Data

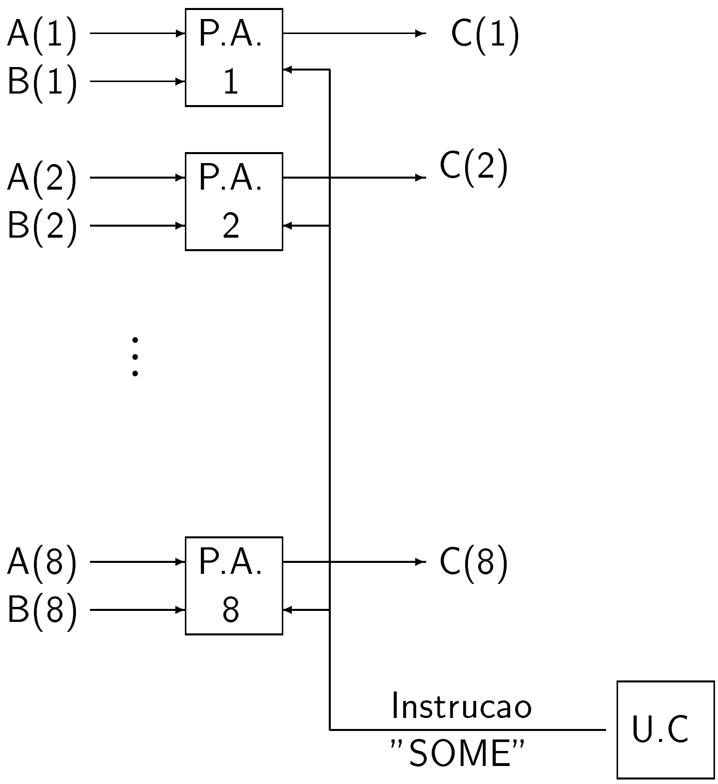


Exemplo: SIMD

Processador Vetorial

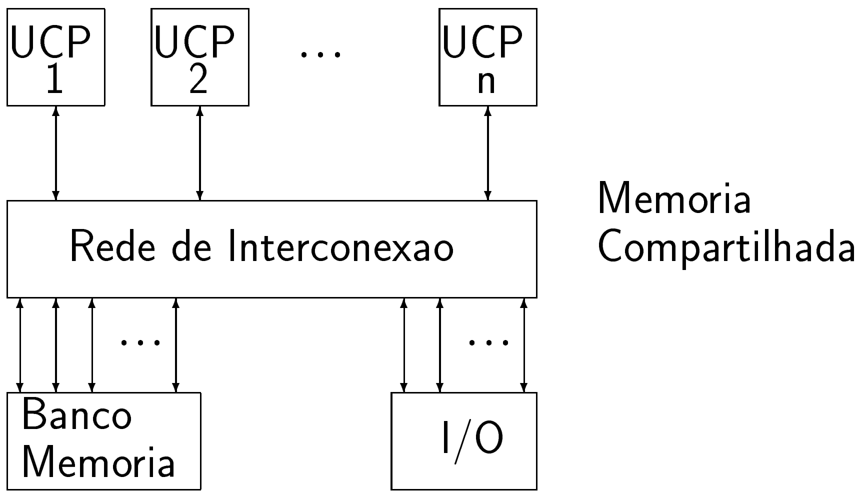
Executa em 1 passo:

$$C(1:8) = A(1:8) + B(1:8)$$

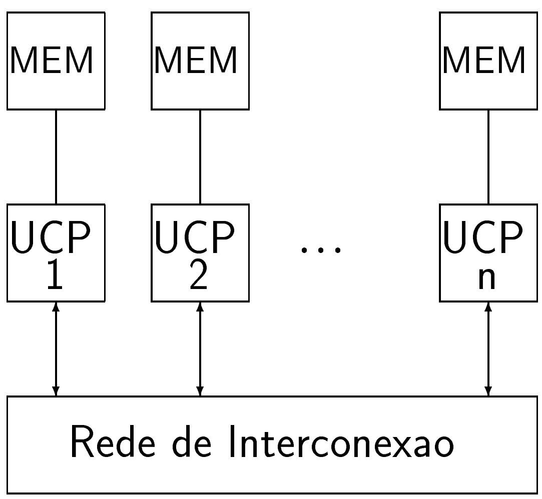


MIMD - Multiple Instruction Multiple Data

Multiprocessador



Multicomputador



MIMD - Comunicação Entre Processadores

Multiprocessadores

Característica Principal:

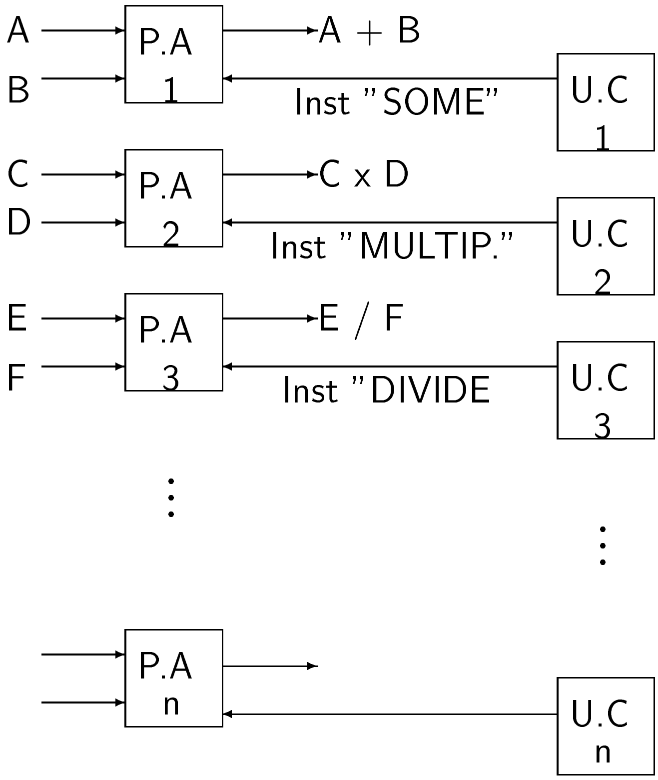
- Memória Compartilhada (através de rede de interconexão centralizada)

Multicomputadores

Característica Principal:

- Comunicação e sincronização via mensagens (toda UCP tem sua própria memória)

Exemplo - MIMD



Speedup

$$S = \frac{\text{Tempo Algoritmo Sequencial usando 1 processador}}{\text{Tempo Algoritmo Paralelo usando p processadores}}$$

Eficiência

$$E = \frac{S}{P}$$

Exemplo:

- Melhor algoritmo sequencial em um comp. paralelo: 8 seg.
- Algoritmo paralelo para o mesmo problema: 2 seg. com 5 proc.

  $\therefore S = 8/2 = 4 \qquad E = 4/5 = 0.8$

Lei de Amdahl

f - Fração Seqüencial

$$0 \leq f \leq 1$$

S - Speedup

$$S \leq \frac{1}{f + \frac{1-f}{p}} = \frac{P}{Pf + (1 - f)}$$

Corolário: Pequeno f limita muito S

Exemplo:

$f = 10\%$

$S \leq 10$ para qualquer número P de processadores!

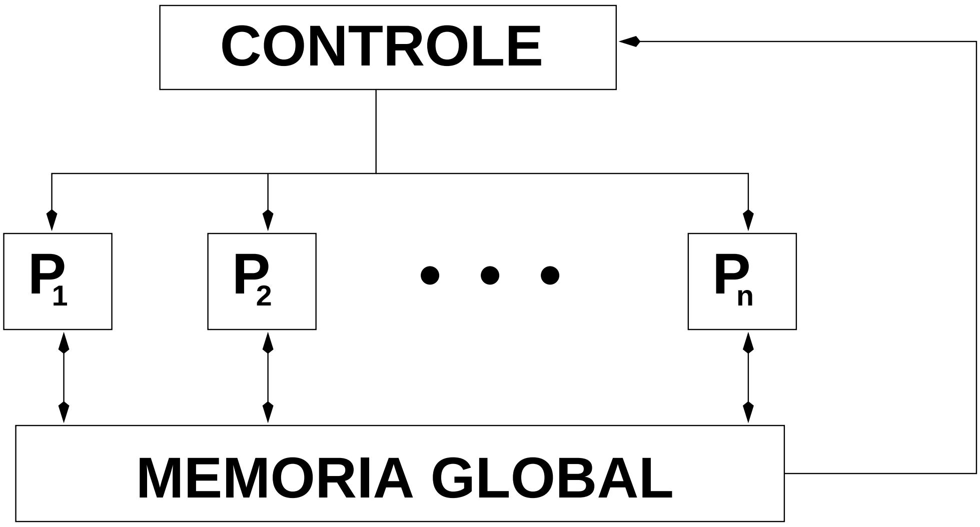
Parallel Random Access Machine

- Poder de processamento ilimitado
- Ignora complexidade de comunicação entre processadores

Custo de uma computação PRAM:

$$\boxed{\text{complexidade paralela} \times p}$$

Modelo Básico PRAM



Cada P_i: Processador + Memória Local

Em um passo da execução: (sincronismo)

- Leitura (local ou global)
- Uma instrução
- Escrita (local ou global)

CREW - Concurrent Read Exclusive Write

Modelos PRAM

Diferem na maneira de lidar com conflitos de E/S

- **EREW (Exclusive Read Exclusive Write)**

  Conflitos de leitura ou escrita não são permitidos

- **CREW (Concurrent Read Exclusive Write)**

  Permite leitura concorrente

  Vários processos podem ler da memória global durante mesma instrução

  Modelo PRAM padrão

Modelos PRAM

- **CRCW (Concurrent Read Concurrent Write)**

  Permite leitura concorrente

  Políticas para lidar com escrita:

  - Comum ("common")

    Processadores escrevem mesmo valor

  - Arbitrário ("arbitrary")

    A escolha é arbitrária

  - Prioritária ("priority")

    Processo com menor índice vence

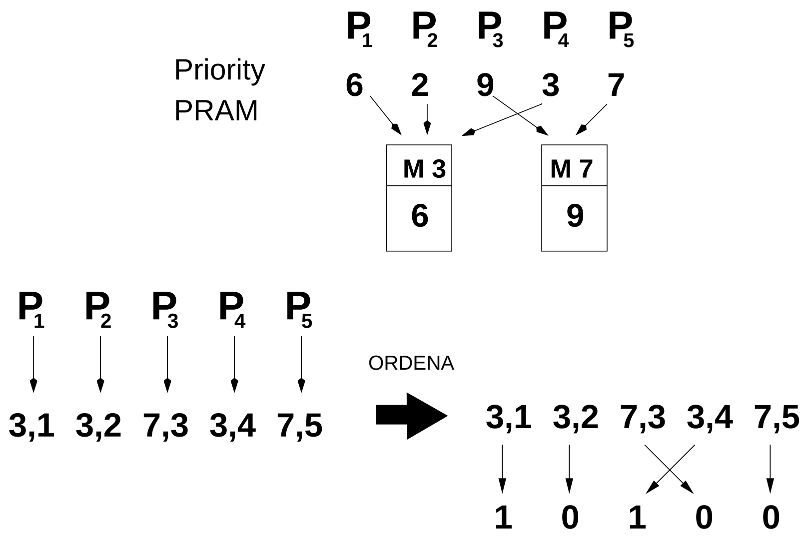
LEMA

Um EREW PRAM com p processadores é capaz de ordenar um arranjo com p elementos armazenado na memória global em $\Theta(\log p)$

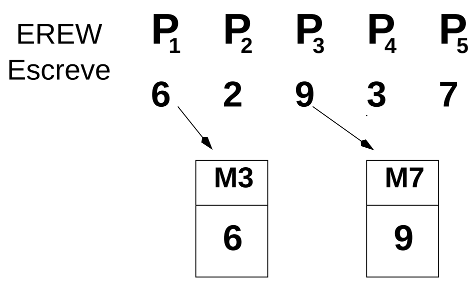
TEOREMA

Um *priority* PRAM com p processadores pode ser simulado por um EREW PRAM com p processadores com uma complexidade de tempo acrescida por um fator de $\Theta(\log p)$

Prova:



Simulação EREW PRAM ($\Theta(\log p)$)



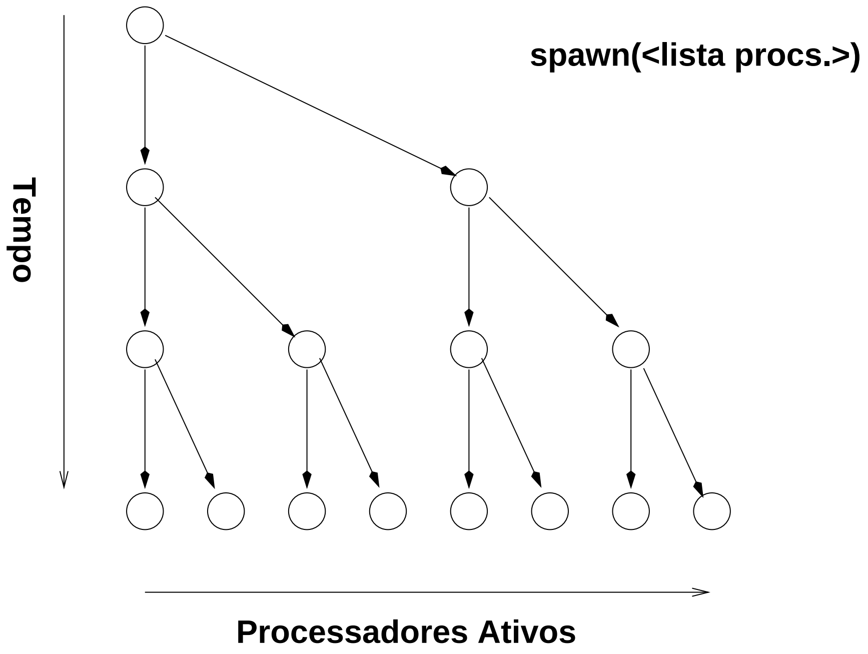
Algoritmos PRAM

Duas fases:

1. Processadores são ativados
2. Processadores ativos realizam computação em paralelo

Na fase 1:

A partir de 1 processador ativo, $\lceil \log p \rceil$ passos são necessários e suficientes para ativar p processadores



Algoritmos PRAM

Na fase 2:

Permite referências a registradores globais

for all $<$*lista procs*$>$ *do*

$<$*lista comandos*$>$ *endfor*

Executa em paralelo todos os comandos listados em cada um dos processadores

Paradigma da Árvore Binária

Fluxo Top-Down

Algoritmos "broadcast"

Raiz envia mesmo dado para folhas

Algoritmos divisão e conquista

Divisão recursiva de problemas em subproblemas

Fluxo Bottom-Up

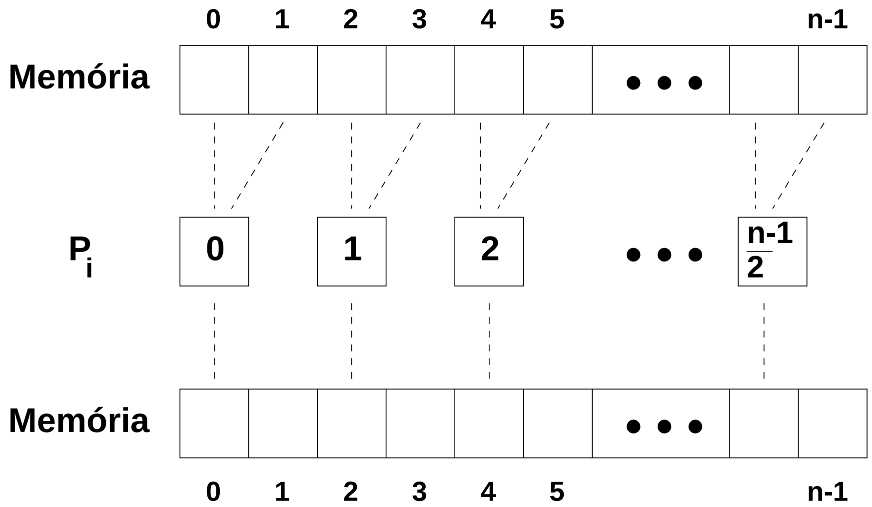
Redução: das folhas para a raiz

Exemplos:

- Máximo de um conjunto
- Soma dos elementos de um conjunto

Soma de um conjunto com n elementos

$n/2$ processadores



Cada processador i compara posições $2i$ e $2i + 1$ e escreve a soma em $2i$

Após primeiro passo $\Rightarrow$ problema de tamanho $n/2$

T(n) $= O(\log n)$

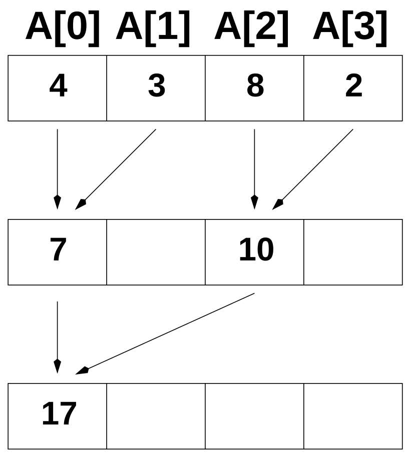
Processadores $= n/2$

Custo $= O(n \log n)$

Soma (EREW PRAM)

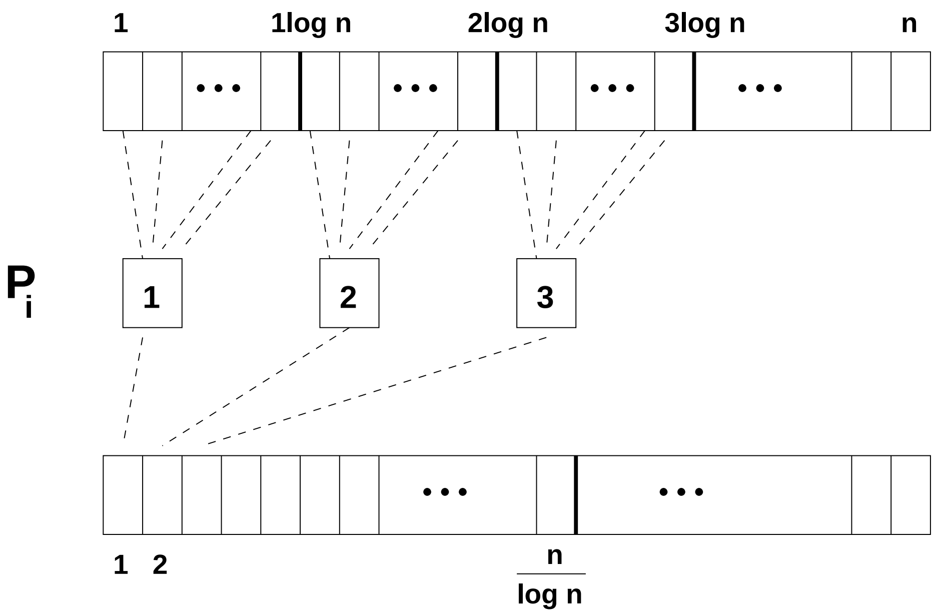
Início: $n \geq 1$ elementos em A[0], A[1], ..., A[$n-1$]
Final: Soma em A[0]
Variáveis Globais: n, A[0..$n-1$], j

> **begin**
>   **spawn** (P_0 , P_1, ..., $P_{\lfloor n/2 \rfloor - 1}$)
>   **for all** P_i **where** $0 \leq i \leq \lfloor n/2 \rfloor - 1$ **do**
>   **for** $j \leftarrow 0$ **to** $\lceil \log n \rceil$-1 **do**
>   **if** $i \; mod \; 2^j = 0$ **and** $2i + 2^j \; < \; n$
>   **then** A[$2i$]$\leftarrow$A[$2i$]+A[$2i$+2^j]
> **end**



Soma Melhorada (EREW PRAM)

Melhoria: $\frac{n}{\log n}$ processadores



Tempo desta fase: $O(\log n)$

Resultam $\frac{n}{\log n}$ números e $\frac{n}{\log n}$ processadores

2ª fase: $O(\log(\frac{n}{\log n}))$, etc.

Total:

$$T(n) = O(\log n)$$

$$\text{Processadores} = \frac{n}{\log n}$$

Custo da Computação Paralela

$\boxed{\text{Tempo} \times \text{Número de Processadores}}$

Algoritmo para obter soma:

T(n) $= O(\log n)$

Processadores $= O(\frac{n}{\log n})$

Produto: $O(\log n) \times O(\frac{n}{\log n}) = O(n)$

Logo, o algoritmo é ótimo.

Algoritmo Ótimo

Definição

Um algoritmo paralelo é ótimo se o seu custo estiver na mesma classe de complexidade do algoritmo sequencial ótimo.

Teorema (Brent, 1974)

Dado um algoritmo paralelo A de custo t, se A executa m operações então p processadores podem executar A em tempo $t + \frac{m-t}{p}$

No caso da soma para $\frac{n}{\log n}$ processadores:

$$\log n + \frac{n-1-\log n}{\frac{n}{\log n}} =$$

$$= \log n + \log n - \frac{\log n}{n} - \frac{\log^2 n}{n} = O(\log n)$$

Prefix Sums

Valores: $a_1, a_2, ..., a_n$

Operação Associativa: $\oplus$

a_1

$a_1 \oplus a_2$

$\ldots$

$a_1 \oplus a_2 \oplus \ldots \oplus a_n$

Exemplo: $+$

$$[3, 1, 0, 4, 2]$$

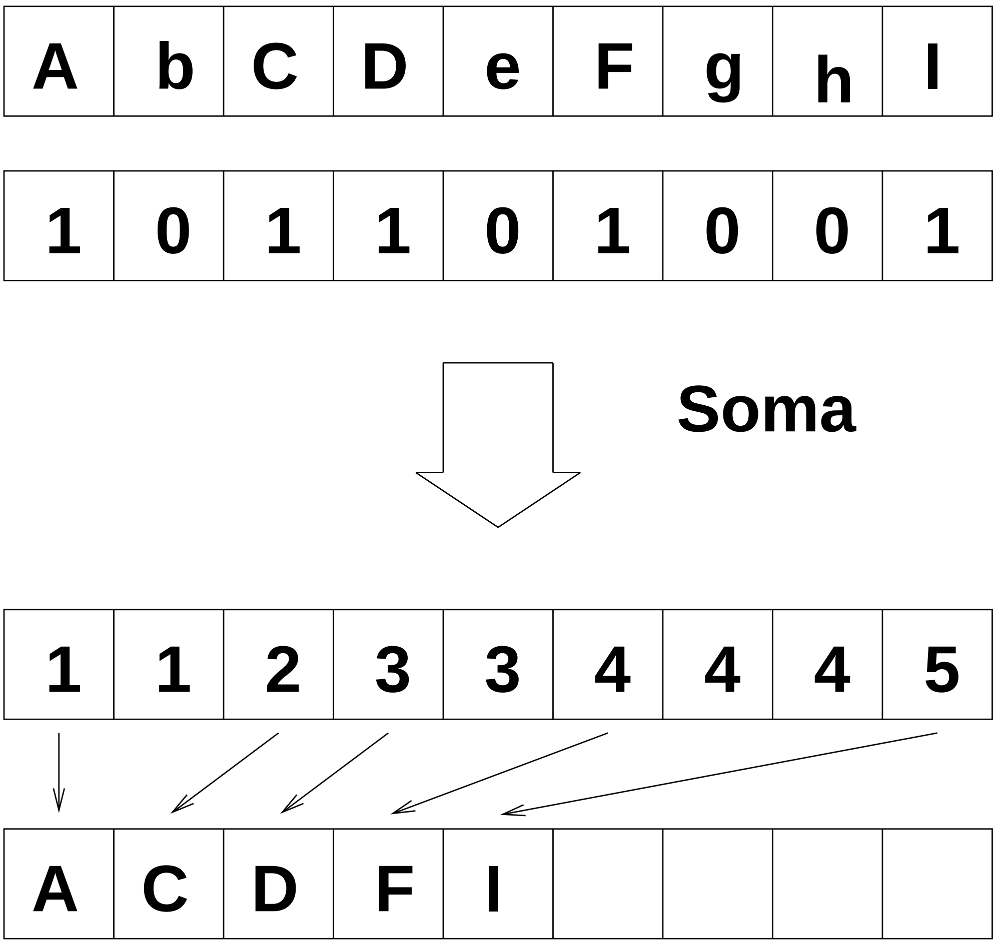
$$\Downarrow$$

$$[3, 4, 4, 8, 10]$$

Prefix Sums

Aplicação: Arranjo A com n letras

Ajuntar maiúsculas mantendo a ordem



Prefix Sums (CREW PRAM)

Início: Lista em A[0..$(n-1)$]

Final: A[i] contém A[0] $\oplus$ A[1] ... $\oplus$ A[n]

Variáveis Globais: n, A[0..$n-1$], j

spawn $(P_1$, P_2, ..., P_{n-1} $)$
for all P_i **where** $1 \leq i \leq n-1$ **do**
    for $j \leftarrow 0$ **to** $\lceil log\ n \rceil$-1 **do**
        if $i - 2^j \geq 0$
        then A[i]$\leftarrow$A[i]$+$A[i-2^j]
    enfor
enfor

Coloração de Grafos

Dado G com n vértices, G: matriz de adjacência $n \times n$

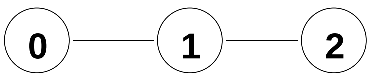
Um processador é criado para cada coloração:

$$P(i_0, i_1, ..., i_n) \Rightarrow \begin{cases} Vértice\ 0 & \rightarrow \quad cor\ i_0 \\ Vértice\ 1 & \rightarrow \quad cor\ i_1 \\ etc. \end{cases}$$

Cada processador verifica se a coloração é válida:

$A[j, k] = 1\ \&\ i_j \neq i_k$ com custo $O(n^2)$

Ex: **G:**



	000	1	0	
	001	1	0	
$c = 2$	011	1	0	
$\Downarrow$	010	1	*1*	$\rightarrow$ 2 colora-
c^n	100	1	0	ções
colorações	101	1	*1*	$\rightarrow$ validas
	110	1	0	
	111	1	0	

Coloração G (CREW PRAM)

Global n, c, j, k
        A$[1..n][1..n]$
        $valid$
begin
    spawn $(P(i_0, i_1, ..., i_{n-1}))$ **where**
        $0 \leq i_v < c$ **for** $0 < v < n$
    for all $(P(i_0, i_1, ..., i_{n-1}))$ **where**
        $0 \leq i_v < c$ **for** $0 < v < n$ **do**
            candidate$[i_0, i_1, ..., i_{n-1}] \leftarrow$ 1
            for $j \leftarrow 0$ **to** $n - 1$ **do**
                for $k \leftarrow 0$ **to** $n - 1$ **do**
                    if A$[j][k]$ **and** $i_j = i_k$
                    then candidate$[i_0, i_1, ..., i_{n-1}] \leftarrow$ 0
                    endif
                endfor
            endfor
            $valid \leftarrow \Sigma$ candidate
    endfor
    if $valid > 0$
    then "Existe coloração valida"
    else "Não existe coloração valida"

Relação entre Modelo PRAM e Teoria de Complexidade

Teorema: (Parallel Computation Thesis)

A classe de problemas que podem ser resolvidos em tempo T$(n)^{O(1)}$ por um computador PRAM é igual a classe de problemas que podem ser resolvidos em espaço T$(n)^{O(1)}$ em um computador RAM, se $T(n) \geq \log n$

Consequência:

PRAM consegue resolver problemas $\mathcal{NP}$-Completo em tempo polinomial, mas usando um número exponencial de processadores

Relação entre Modelo PRAM e Teoria de Complexidade

Complexidade poli-logarítmica

Para entrada de tamanho n pior caso $T(n)$ é um polinômio do logaritmo de n (*poly-log-time*)

Classe $\mathcal{NC}$

Problemas que podem ser resolvidos em paralelo em tempo $O(\log^k n)$, $k = O(1)$ e com um número polinomial de processadores

$\mathcal{NC}$ - Nick's Class (Nickolas Pippenger)

Problemas P-Completo

Existem muitos problemas em $\mathcal{P}$ com solução *poly-log-time* em paralelo

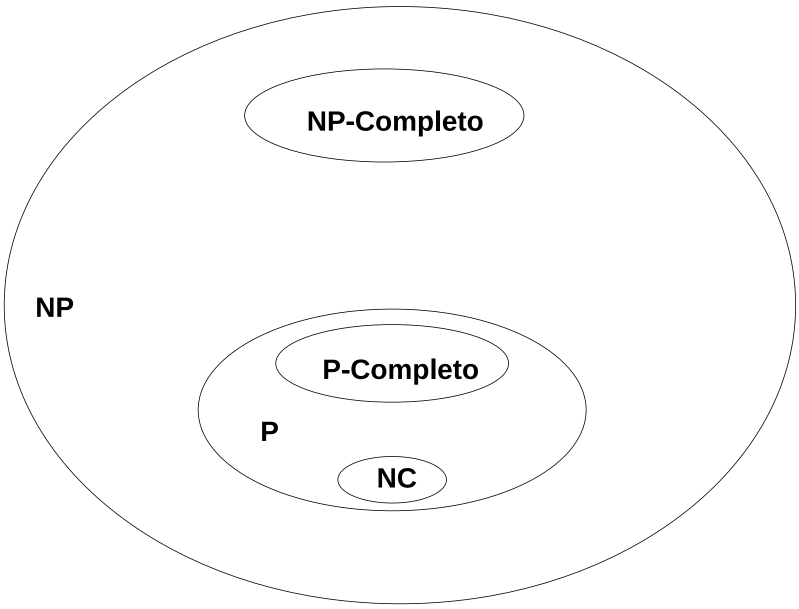
Entretanto, está em aberto se $\mathcal{NC} = \mathcal{P}$

Definição

Um problema $\Pi \in \mathcal{P}$ é $\mathcal{P}$-Completo se qualquer outro problema em $\mathcal{P}$-Completo puder ser transformado para Π em tempo poli-logarítmico usando PRAM com um número polinomial de processadores

Problemas $\mathcal{P}$-Completo

São problemas que parecem não existir uma solução paralela eficiente (poli-logarítmica)



Descrição tentativa do mundo

Em aberto $\begin{cases} \mathcal{NC} = \mathcal{P} \\ \mathcal{P} = \mathcal{NP} \end{cases}$

$1^{\underline{o}}$ Problema $\mathcal{P}$-Completo

Problema da Generalidade

Instância: Um conjunto X e um operador binário $\bullet$

Questão: $x \in X$ pertence ao fechamento ("closure") com respeito a $\bullet$ de T $\subseteq X$?

Exemplos de problemas $\mathcal{P}$-Completo:

1. Dado um circuito booleano com um conjunto de valores como entrada.

   Questão: A saída é verdadeira?

   (CVP - Circuit Value Problem)

2. Depth-First-Search

   Dado um grafo $G = (V, A)$, visitar todos os vértices a partir de v_0

Projeto de Algoritmos Paralelos

- Explorar paralelismo inerente de algoritmo seqüencial existente.
- Inventar um novo algoritmo paralelo
- Adaptar outro algoritmo paralelo que resolva problema similar

Projeto de Algoritmos Paralelos

Conhecimento do problema é muito importante.

Um algorimo seqüencial bem conhecido pode ajudar:

SOMA; {SISD}
begin
    $s \leftarrow a_0$
    for $i \leftarrow 1$ **to** $n - 1$ **do**
        $S \leftarrow S + a_i$
end

Para $n = 4$: $[(a_0 + a_1) + a_2] + a_3$
este algoritmo é paralelizável?

Sim: $(a_0 + a_1) + (a_2 + a_3)$ - propriedade aditiva da adição

Projeto de Algoritmos Paralelos

Custos de comunicação têm que ser considerados

Algumas ocasiões

complexidade comunicação $>$ complexidade de tempo

Exemplo:

custo 1 adição ponto flutuante: i

custo passar 1 número p.f. entre processadores: $100i$

Problema: Somar n números p.f. usando P processadores

1° passo: distribuir n pelos P processadores

se $n \leq 101$

então mais rápido somar em 1 processador

Independente do número P, porque custo de 100 adições é maior que o de 1 comunicação.

Projeto de Algoritmos Paralelos

Algoritmo deve considerar arquitetura

Considerações importantes:

- Comunicação (como visto antes)
- Sincronização
  - Modelo SIMD: automática (eficiente)
  - Modelo MIMD: software (consome tempo)

    Logo, algoritmos com poucas operações entre sincronizações $\Rightarrow$ Baixa eficiência em computadores MIMD

Algoritmos para máquinas SIMD

- Sincronização: realizada pela arquitetura
- Comunicação: custo é significativo

Terminologia

p: número de processadores

P_i: processador i, $0 \leq i < p$

for all .. end for: ativa conjunto de processadores

$\Leftarrow$ comunicação de um item de dado entre processadores vizinhos (de um processador vizinho para um processador "ativo")

Estratégias para Projeto de Algoritmos Paralelos

Estratégia 1

Se existe um algoritmo PRAM CREW ótimo e a forma de interação entre processadores através de variáveis compartilhadas mapeia na arquitetura, então o algoritmo PRAM é um bom ponto de partida.

Objetivo

Introduzir o menor número possível de operações extras no algoritmo paralelo comparado ao melhor algoritmo sequencial

Em outras palavras

Se as constantes de proporcionalidade do algoritmo PRAM ótimo e o algoritmo sequencial são próximas então é possível usar o algoritmo paralelo PRAM como ponto de partida

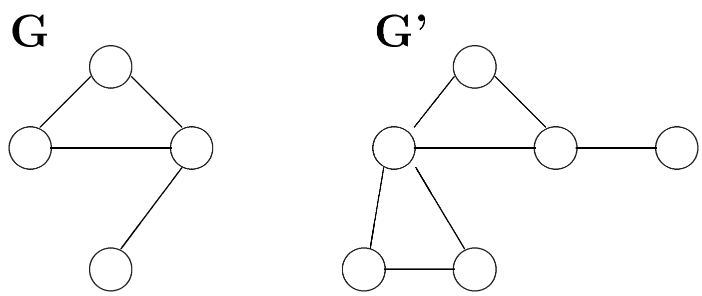
Dilação ("Dilation")

Seja ϕ função que encaixa $G = (E, V)$ dentro de $G' = (E', V')$.

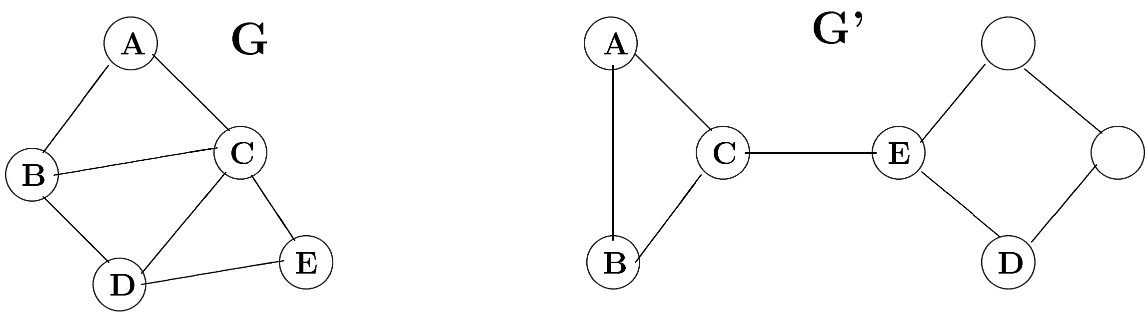
A **dilação** do encaixe é: dil(ϕ)=$max\{dist(\phi(u), \phi(v)) | (u, v) \in E\}$

onde $dist(a, b)$ é a distância entre os vértices a e b em G'

Exemplos:



Dilação-1: Se G subgrafo de G' então existe uma dilação-1



Dilação-3 do encaixe de G em G'
(arco (B, D) em $G \rightarrow$ caminho de comprimento 3 em G')

Soma de um conjunto

Existe um algoritmo PRAM ótimo: $n/\log n$ processadores podem adicionar n elementos em $\Theta(\log n)$

Logo:
Podemos usar o mesmo princípio para desenvolver bons algoritmos paralelos para máquinas SIMD e MIMD, mesmo que $p < n/\log n$

Importante:
Constante do algoritmo PRAM não deve ser muito maior do que constante do algoritmo sequêncial.

Se interação entre processadores PRAM forma um grafo que encaixa com dilação-1 em uma arquitetura SIMD então existe uma tradução natural do algoritmo PRAM para o algoritmo SIMD.

Soma de um conjunto

Como descobrir se existe ou não uma boa transformação de um algoritmo PRAM para uma topologia qualquer?

(No caso da adição no SIMD-PS não existe encaixe com dilação-1 e ainda assim existe algoritmo eficiente.)

Para evitar uma possível procura inútil:

- estabelecer limite inferior para qualquer algoritmo na topologia em questão.

Adição no Modelo SIMD - MC^2

Somar $n = l \times l$ valores

Processador P_{ji} processa variáveis locais t_{ji} e a_{ji} para todo i, j, $\begin{cases} 1 \leq i \leq l \\ 1 \leq j \leq l \end{cases}$

adição; $\{$SIMD - MC$^2\}$
begin
  for $i \leftarrow l - 1$ **downto** 1 **do**
    for all $P_{j,i}, 1 \leq j \leq l$ **do** $\{$col. i ativa$\}$
      $t_{j,i} \Leftarrow a_{j,i+1}$
      $a_{j,i} \leftarrow a_{j,i} + t_{j,i}$
    end for
  end for
  for $i \leftarrow l - 1$ **downto** 1 **do**
    for all $P_{i,1}$ **do** $\{$1 proc. ativo$\}$
      $t_{i,1} \Leftarrow a_{i+1,1}$
      $a_{i,1} \leftarrow a_{i,1} + t_{i,1}$
    end for
  end for
end

Adição no Modelo SIMD - MC^2

$n = 16$ SIMD - MC^2

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

1o FOR

0	1	5	-
4	5	13	-
8	9	21	-
12	13	29	-

1a Adicão

0	6	-	-
4	18	-	-
8	30	-	-
12	42	-	-

2a Adicão

6	-	-	-
22	-	-	-
38	-	-	-
54	-	-	-

3a Adicão

2o FOR

6	-	-	-
22	-	-	-
92	-	-	-
-	-	-	-

4a Adicão

6	-	-	-
114	-	-	-
-	-	-	-
-	-	-	-

5a Adicão

120	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-

6a Adicão

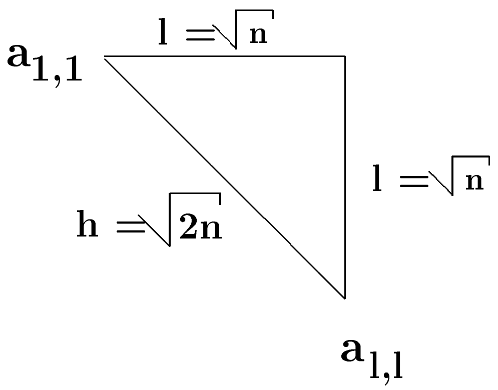
Complexidade

1º Anel Externo: $l - 1$
2º Anel Externo: $l - 1$
Cada *for all*: $O(1)$

$$T(n) = 2(l-1) = 2\sqrt{n} - 2$$

$$T(n) = O(\sqrt{n})$$

Adição no SIMD-MC^2 pode ser melhorada?
Resposta: NÃO Porquê?



Na realidade, precisamos de $2\sqrt{n} - 2$ roteamentos para adicionar $a_{1,1}$ e $a_{l,l}$.

Estratégias para Projeto de Algoritmos Paralelos

Estratégia 2

Procure por um algoritmo com paralelismo de dados antes de considerar um algoritmo com paralelismo de controle.

SIMD: apenas paralelismo de dados

$$\text{MIMD: } \begin{cases} \text{Paralelismo de dados} \\ \text{Paralelismo de controle} \end{cases}$$

Algoritmos com paralelismo de dados:

- Mais comum
- Mais fáceis de projetar e depurar
- Melhores para ampliar número de processadores

Algoritmos para Máquinas MIMD

Questões relativas a multiprocessadores ou multicomputadores

- Como dividir o trabalho entre processadores?
- Quais mecanismos permitem o trabalho em conjunto de processadores?
- Como alocar tarefas nos processadores?

Computadores MIMD

- Mais gerais
- Execução assíncrona de múltiplas instruções
- Maior flexibilidade no projeto de algoritmos

Algoritmos MIMD

Particionar corresponde a dividir uma computação:

- Problema é dividido em subproblemas que são resolvidos por processadores individuais.
- As soluções dos subproblemas são combinadas.
- Para combinar as soluções: $\Rightarrow$ sincronização.

Comunicação e sincronização de processos (MIMD)

Comunicação

- Variáveis compartilhadas (multiprocessadores)
- Via mensagem (multicomputadores)

Sincronização

- Controla ordem de eventos
- Controla interferência

Classificação de Algoritmos MIMD

Paralelismo de dados:

Mesma operação aplicada simultaneamente a elementos de um conjunto de dados

Pré-planejado ("prescheduled"):

Número de itens de dados por unidade funcional é determinado antes do início do processamento

Auto-Planejado ("self-scheduled"):

Itens de dados não são atribuídos a unidades funcionais antes do início do processamento

Algoritmos MIMD

Prescheduled

O pedaço de cada processo é alocado em tempo de compilação.

Está no código a decisão de que cada processo deve adicionar k valores

Self-scheduled

O trabalho é atribuído a cada processo em tempo de execução.

Uma lista global de tarefas é mantida.

Processo inativo busca nova tarefa na lista.

Algoritmos MIMD: Multiprocessador

Ex: Adicionar K_p valores em um multiprocessador contendo p processadores.

- *variavel global* $\leftarrow 0$
- Cria p processos (1 por processador)
- Cada processo adiciona k valores
- Quando um processo obtém seu subtotal, adiciona à *variavel global*.

$\Rightarrow$ processo tem que ter acesso exclusivo a *variavel global*

Algoritmos MIMD: Multiprocessador

Memória Global

Valores a serem adicionados:
$a_0, a_1, ..., a_{n-1}$

$Global\_sum$: resultado final

Cada processador P_i:

variável local j_i: índice do loop

$Local\_sum$: subtotal do processador

for all: é executado por p processos de forma assíncrona

Lock
Unlock: funções que implementam uma seção crítica.

(*locking* é uma ação indivisível: nenhum outro processo pode bloquear uma *"locked variable"*).

Divisão do trabalho (MIMD)

Algoritmo multiprocessadores para adição

adição; {MIMD - memoria compartilhada}
begin
  $Global\_sum \leftarrow 0$;
  for all $P_i, 0 \leq i < p - 1$ **do**
    $Local\_sum \leftarrow 0$;
    for $j \leftarrow i$ **to** $n - 1$ **step** p **do**
      $Local\_sum \leftarrow Local\_sum + a_j$;
    end for;
    lock($Global\_sum$);
      $Global\_sum \leftarrow Global\_sum + Local\_sum$;
    unlock($Global\_sum$);
  end for
end

Complexidade do Pior Caso

Criação dos p processos: $O(p)$

$Local\_sum$: $O(n/p)$

Não há conflito de memória:
p processadores

p bancos de memória

$Global\_sum$: $O(p)$

Atualizada dentro de uma seção crítica

Complexidade final: $O(n/p + p)$

Mínimo de $n/p + p$ ocorre quando $p = \sqrt{n}$

Logo, tempo é minimizado quando $p = \sqrt{n}$

Algoritmos MIMD: Assíncronos

- Trabalham sem sincronização de processos
- Nenhum processo necessita esperar dados de outro processo
- Processadores trabalham com dados mais recentemente disponíveis
- Comportamento não determinístico

Como expressar concorrência

Fork e *Join*

program a;	program b;
...	...
fork b;	...
...	...
join b;	end;
...	

- Um processo executa $\underline{a}$ até encontrar o comando *fork*
- Neste momento, $\underline{b}$ inicia equanto $\underline{a}$ continua
- Se o processo executando a encontra o *join* $\underline{b}$ antes de $\underline{b}$ terminar, então execução do primeiro processo é suspensa até o término de $\underline{b}$

Sincronização com variáveis compartilhadas

Seção Crítica

Seqüência de comandos que devem ser executados como uma operação indivisível.

Exclusão Mútua

Refere-se a execução exclusiva de seções críticas

Semáforos (para implementar sincronização)

Inteiro não negativo que só pode ser modificado por:

wait(s): when $s > 0$ *do* $s := s - 1$;

signal(s): $s := s + 1$;

Processo executando *wait(s)* é atrasado até que $s > 0$, quando então faz $s := s - 1$ e continua

wait e *signal* são operações indivisíveis.

Deadlock

Ocorre quando um conjunto de processadores ativos mantém recursos que são necessários a outros processos.

Ex:

processo 1	processo 2
...	...
lock(A)	*lock*(B)
...	...
lock(B)	*lock*(A)

lock, unlock correspondentes a *wait, signal*

Se nenhum dos dois processos for "forçado" a liberar seu semáforo então eles ficarão travados por tempo indefinido.