Fontes principais

1. J. Jaja, An introduction to Parallel Algorithms, Addison Wesley, 92

   ▷ Algoritmos paralelos

2. E. Cáceres, H. Mongeli, S. Song: Algoritmos paralelos usando CGM/PVM/MPI: uma introdução
   `http://www.ime.usp.br/~song/papers/jai01.pdf`

Por que discutir Computação Paralela?

- Buscar execuções cada vez mais rápidas dos programas.
- Processamento sobre grande volume de dados

Sistemas de computação paralela

Um **sistema de computação paralela** é uma coleção de processadores interconectados de maneira a permitir a coordenação de suas atividades e a troca de dados.

Os processadores trabalham simultaneamente, de forma coordenada para resolver um problema.

Aplicações do Paralelismo

▷ Problemas computacionalmente custosos, que demandam muito tempo de processamento em computadores sequenciais.

▷ Problemas com entrada de dados muito grande

▷ Problema que não possuem algoritmos sequenciais eficientes também não terão algoritmos paralelos eficientes. Problemas que possuem algoritmos sequenciais eficientes podem ter ou não algoritmos paralelos eficientes

Classificação de Flynn (1966)

Os sistemas de computação paralela podem ser definidos de acordo com:

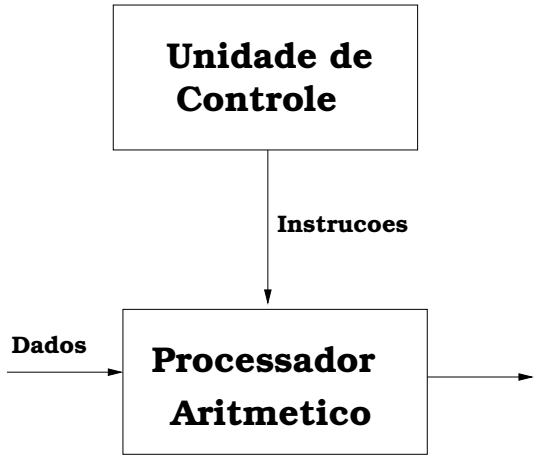
- Fluxo de instruções
- Fluxo de dados

	SD (Single Data)	MD (Multiple Data)
SI (Single Instruction)	SISD	SIMD
MI (Multiple Instruction)	MISD	MIMD

SISD - Single Instruction Single Data

Computador de Von Newman

- Computadores convencionais
- Uma CPU e uma unidade de controle conectadas por barramento.



SIMD - Single Instruction Multiple Data

Vários processadores homogêneos com acesso à memória compartilhada em tempo constante.

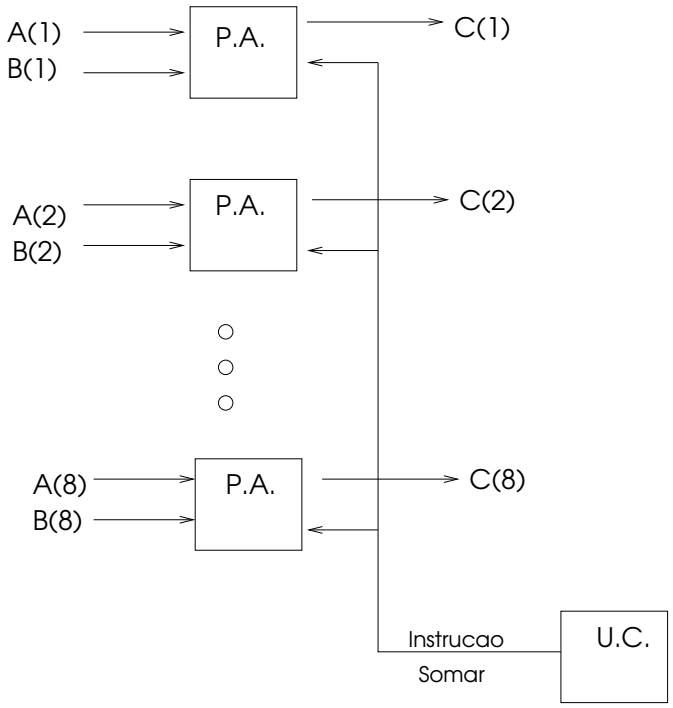
Mesma instrução executando sobre dados distintos

Ex.: Processador vetorial

Processador vetorial

Executa em 1 passo

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

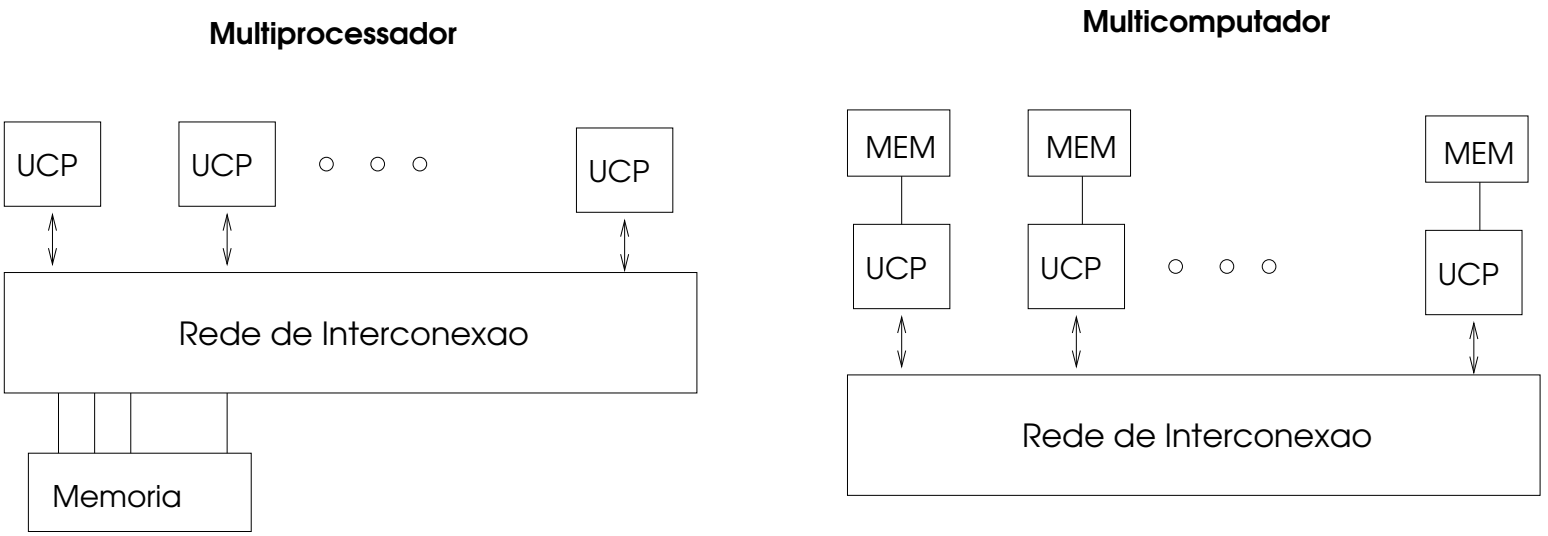


MIMD - Multiple Instruction Multiple Data

Cada processador executa o seu próprio fluxo de instruções.

- Multicomputadores
- Multiprocessadores

MIMD - Multiple Instruction Multiple Data



Sistemas de computação paralela

Existem outras formas de classificar os sistemas de computação paralela, entre eles, destacamos: **dispersão de computadores**, **estruturas de interconexão**, e **sincronismo**.

Modelos de computação paralela

Modelos de computação paralela

Existem vários tipos de arquiteturas paralelas que implementam diferentes sistemas de computação paralela. Para cada arquitetura paralela, temos modelos distintos de desenvolvimento de algoritmos paralelos.

Estes modelos, nos quais baseamos o desenvolvimento de algoritmos, são denominados **modelos de computação paralela e distribuída**.

Modelos de computação paralela

Memória compartilhada

- PRAM

Modelo de rede

- Array, Anel, Hipercubo, Malha, Torus, Estrela

Realístico

- BSP, CGM

Modelos PRAM

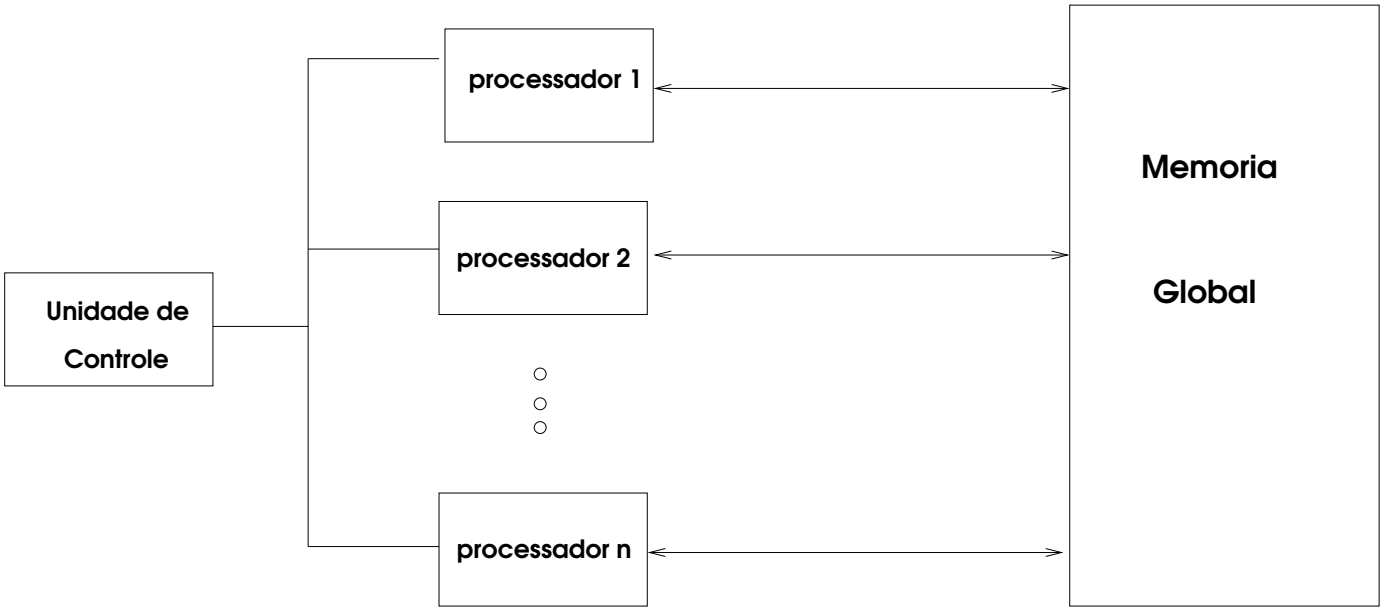
Nos algoritmos sequenciais o modelo **RAM (Random Access Machine)** é bastante próximo da forma de escrever algoritmos em máquinas Von Neuman.

O modelo **PRAM (Parallel Random Access Machine)** é uma extensão do modelo RAM.

Modelos PRAM

- O modelo PRAM consiste de um conjunto de processadores conectados a uma memória global.
- Existem n processadores ligados a uma memória global, e cada processador é identificado por um número inteiro.
- A memória global pode ser acessado por qualquer processador.

Modelos PRAM



Modelos PRAM

- Existe uma única unidade de controle, que passa a instrução a ser executada para todos os processadores (SIMD).
- Em cada instante, todos os processadores ativos executam a mesma instrução sobre dados possivelmente diferente.
- Existe um único relógio (clock) global compartilhado pelos processadores. Modelo síncrono.
- Processadores se comunicam através da memória compartilhada, utilizando variáveis compartilhadas.

Exemplos

Algoritmos paralelos no modelo PRAM

para $1 \leq i \leq n-1$ faça em paralelo
  $C[i] := A[i] + B[i]$
  $F[i] := D[i] + E[i]$
fim para

para $1 \leq i \leq n-1$ faça em paralelo
  $G[i] := C[i+1]$
fim para

Modelo de Computação Paralela e Distribuída

- Modelo MIMD.
- Existem p processadores que executam em paralelo e estão interligados através de uma rede de interconexão.
- Cada processador é identificado por um número inteiro

Modelo de Computação Paralela e Distribuída

- Cada processador possui associado a ele uma memória local, acessível apenas a ele. Um processador não possui acesso à memória local associada aoutro processador (memória distribuída).

- Cada posição de memórias locais possui um endereço e pode ser acessada apenas por seu processador associado.

- A cada instante os processadores podem estar ativos ou inativos.

Modelo de Computação Paralela e Distribuída

- Cada processador possui sua unidade de controle, que passa a instrução a ser executada para ele.

- Em cada instante, cada processador ativo executa uma instrução, possivelmente diferente dos demais, sobre os dados possivelmente diferentes (MIMD)

- Não existe relógio global. Cada processador possui o seu relógio local. Modelo assíncrono.

Modelo de Computação Paralela e Distribuída

- Mesmo que os processadores ativos estejam executando a mesma sequência de instruções, eles estarão executando a taxas diferentes.

- Processadores se comunicam através da rede de interconexão, usando troca de mensagens.

Exemplos

Processador 1

para $1 \leq i \leq n-1$ faça
  $C[i] := A[i] + B[i]$
  se $i \neq 1$ então
    $envia(proc3, C[i])$
  fim se
fim para

Processador 2

para $1 \leq i \leq n-1$ faça
  $F[i] := D[i] + E[i]$
fim para

Processador 3

para $1 \leq i \leq n-1$ faça
  $recebe(M)$
  $G[i] := M$
fim para

Comparações entre modelos

Modelo PRAM

- Execução síncrona
- Memória compartilhada
- Comunicação através de memória compartilhada

Modelo Distribuído

- Execução assíncrona
- Memória distribuída
- Comunicação através de troca de mensagens

Modelo de Computação Paralela e Distribuída

- Se um processador i precisar de um dado que o processador j calculou, para faze um novo cálculo, o processador j envia este dado para o processador i (em uma mensagem), através da rede, e o processador i recebe este dado.

- Os termos de transmissão de mensagens através da rede são indeterminados (porém finitos).

Códigos

Algoritmo Kruskal(G, p)
$T \leftarrow \emptyset$
para cada vértice $v \in V(G)$
    Associe um conjunto a v
fim para
E arestas em $A(G)$ ordenadas, pelo peso, de forma não-decrescente.

enquanto $E \neq \emptyset$
    $uv \leftarrow$ ExtraiPrimeiro(E)
    se conjunto associado a $u \neq$ conjunto associado a v então
        Agrupe u e v em um mesmo conjunto
        $T \leftarrow T \cup uv$
    fim se
fim enquanto
retorne $(V(G), T)$

Códigos

```
This code is from "Algorithms in C, Third Edition,"
by Robert Sedgewick, Addison Wesley Longman, 1998.
```

Problema da conexidade

```
yoshi@erdos:~/Main/www/2006ii/mac122a/exx$ cat prog1.1.in
3 4
4 9
8 0
2 3
5 6
2 9
5 9
7 3
4 8
5 6
0 2
6 1
yoshi@erdos:~/Main/www/2006ii/mac122a/exx$
```

Problema da conexidade

```
yoshi@erdos:~/Main/www/2006ii/mac122a/exx$ prog1.1 < prog1.1.in
 3 4
 4 9
 8 0
 2 3
 5 6
 5 9
 7 3
 4 8
 6 1
yoshi@erdos:~/Main/www/2006ii/mac122a/exx$
```

Programas para o problema da conexidade

```
/* prog1.1.c - Quick find */
#include <stdio.h>
#define N 10000
main()
  { int i, p, q, t, id[N];
    for (i = 0; i < N; i++) id[i] = i;
    while (scanf("%d %d\n", &p, &q) == 2)
      {
        if (id[p] == id[q]) continue;
        for (t = id[p], i = 0; i < N; i++)
          if (id[i] == t) id[i] = id[q];
        printf(" %d %d\n", p, q);
      }
 }
```

Quick find

Propriedade 1. *Suponha que executamos o algoritmo quick find em uma instância com M pares e N objetos, e que a saída tem S pares. Então o algoritmo executou pelo menos NS instruções (por exemplo, ele executou o teste* `id[i] == t` *pelo menos este número de vezes).*

▷Note que S pode chegar a ser $N - 1$. Assim, quick find pode ter tempo de execução tão grande quanto $N(N - 1)$ unidades.

```
/* prog1.2.c - Quick union */
#include <stdio.h>
#define N 10000
main()
  { int i, p, q, t, id[N];
    for (i = 0; i < N; i++) id[i] = i;
    while (scanf("%d %d\n", &p, &q) == 2) {
      for (i = p; i != id[i]; i = id[i]) ;
      for (j = q; j != id[j]; j = id[j]) ;
      if (i == j) continue;
      id[i] = j;
      printf(" %d %d\n", p, q);
    }
  }
```

Quick union

Propriedade 2. *O algoritmo quick union pode chegar a executar* $(N-1)(N-2)/2$ *instruções para resolver o problema da conexidade com* $N-1$ *pares e* N *objetos.*

▷ Para verificar a propriedade acima, considere a entrada cujos pares são $(0,1),(0,2),\ldots,(0,N-1)$. Conte o número de vezes que a instrução `i = id[i]` (no `for`) é executada.

```
/* prog1.3.c - Weighted quick union */
#include <stdio.h>
#define N 10000
main()
  { int i, j, p, q, id[N], sz[N];
    for (i = 0; i < N; i++)
      { id[i] = i; sz[i] = 1; }
    while (scanf("%d %d\n", &p, &q) == 2)
      { for (i = p; i != id[i]; i = id[i]) ;
        for (j = q; j != id[j]; j = id[j]) ;
        if (i == j) continue;
        if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
        else { id[j] = i; sz[i] += sz[j]; }
        printf(" %d %d\n", p, q);
      }
  }
```

Quick union com pesos

Propriedade 3 (Custo do find)**.** *O algoritmo quick union com pesos segue* $\leq \log_2 N$ *ponteiros para chegar à raiz da árvore que contém o elemento de partida.*

Prova. Basta provar o seguinte fato sobre as árvores que ocorrem em nossa estrutura de dados: se uma árvore tem k elementos, então ela tem altura no máximo $\log_2 k$. Suponha o fato verdadeiro em um dado momento. Suponha que unimos duas árvores por uma operação de *union*. Suponha que as árvores tinham i e i' elementos, com $i \leq i'$, e alturas h e h'. Se $h < h'$, então a nova árvore tem altura h'. Se $h = h'$, então a nova árvore tem altura $h + 1 \leq \log_2 i + 1 = \log_2(i + i) \leq \log_2(i + i')$. □

Quick union com pesos

Corolário 4. *O algoritmo quick union com pesos demora tempo não mais que proporcional a* $M \log N$ *para resolver o problema da conexidade com* M *pares e* N *objetos.*

- ▷ **Dizemos:** a complexidade de tempo do quick union com pesos é $O(M \log N)$.

Quick union com compressão de caminhos

É possível melhorar o quick union com pesos, usando *compressão de caminhos*, como no programa a seguir. [Não chegamos lá na aula.]

```
/* prog1.4.c - Weighted quick union with path compression */
#include <stdio.h>
#define N 10000
main()
  { int i, j, p, q, id[N], sz[N];
    for (i = 0; i < N; i++) { id[i] = i; sz[i] = 1; }
    while (scanf("%d %d\n", &p, &q) == 2)
      { for (i = p; i != id[i]; )
          { int t = i; i = id[id[t]]; id[t] = i; }
        for (j = q; j != id[j]; )
          { int t = j; j = id[id[t]]; id[t] = j; }
        if (i == j) continue;
        if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
        else { id[j] = i; sz[i] += sz[j]; }
        printf(" %d %d\n", p, q);
      }
  }
```

Exercício

▷ **Instância conexa:** dizemos que uma instância é conexa quando a saída tem $N - 1$ pares.

Para contar o número de pares na saída, basta fazer

```
yoshi@RANDOM ~/Main/www/2006ii/mac122a/exx
$ prog1.1 < prog1.1.in | wc -l
9
yoshi@RANDOM ~/Main/www/2006ii/mac122a/exx
$
```

Verifique experimentalmente a *probabilidade de uma instância aleatória ser conexa* para valores grandes de N e M por volta de $N(\log N)/2$. (Você tem de escrever um pequeno programa para gerar as instâncias aleatórias.)