Aula 02 – Software Paralelo

Prof. Paulo Bressan

Conceitos Básicos

- Balanceamento de Carga
  - Capacidade de distribuir tarefas pelos processadores
  - De forma que todos estejam ocupados todo o tempo.

Estático: em tempo de compilação

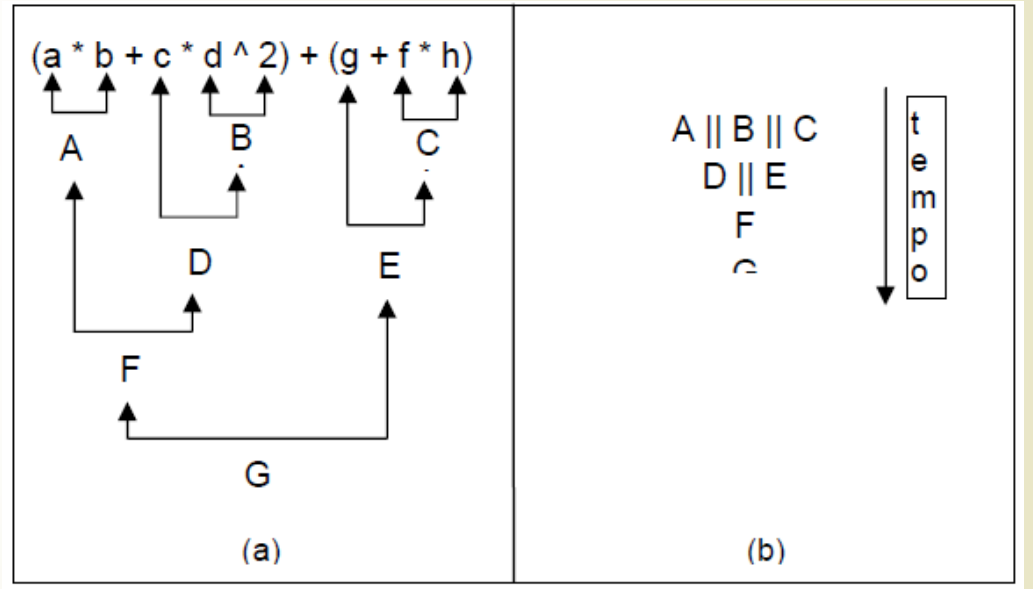
Dinâmico: em tempo de execução

Paralelização Automática

- Seja a expressão:

(a * b + c * d ^ 2) + (g + f * h).

- Uma possível organização da execução de cada uma das operações, respeitando-se a precedência, é:

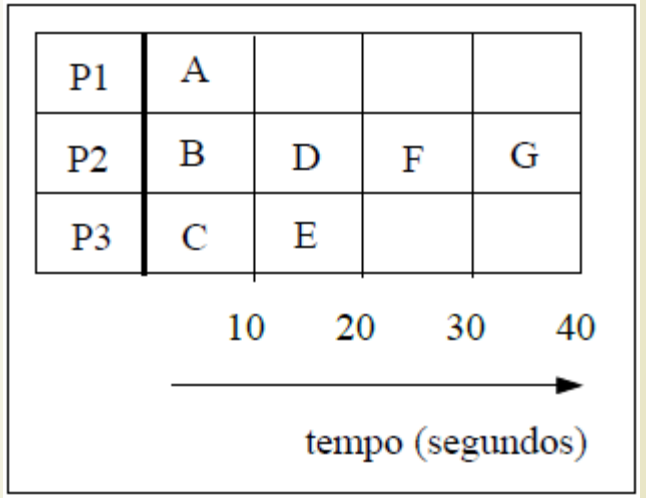


Paralelização Automática

- Cada tarefa (A, B, C, ..., G) executa uma operação matemática entre dois números.
- Pode-se organizar a execução de cada tarefa através do diagrama apresentado em b.
- Supondo-se que se possua três processadores (P1, P2 e P3), pode-se representar a distribuição das tarefas entre eles através de um diagrama *processadores X tempo*, apresentado a seguir.

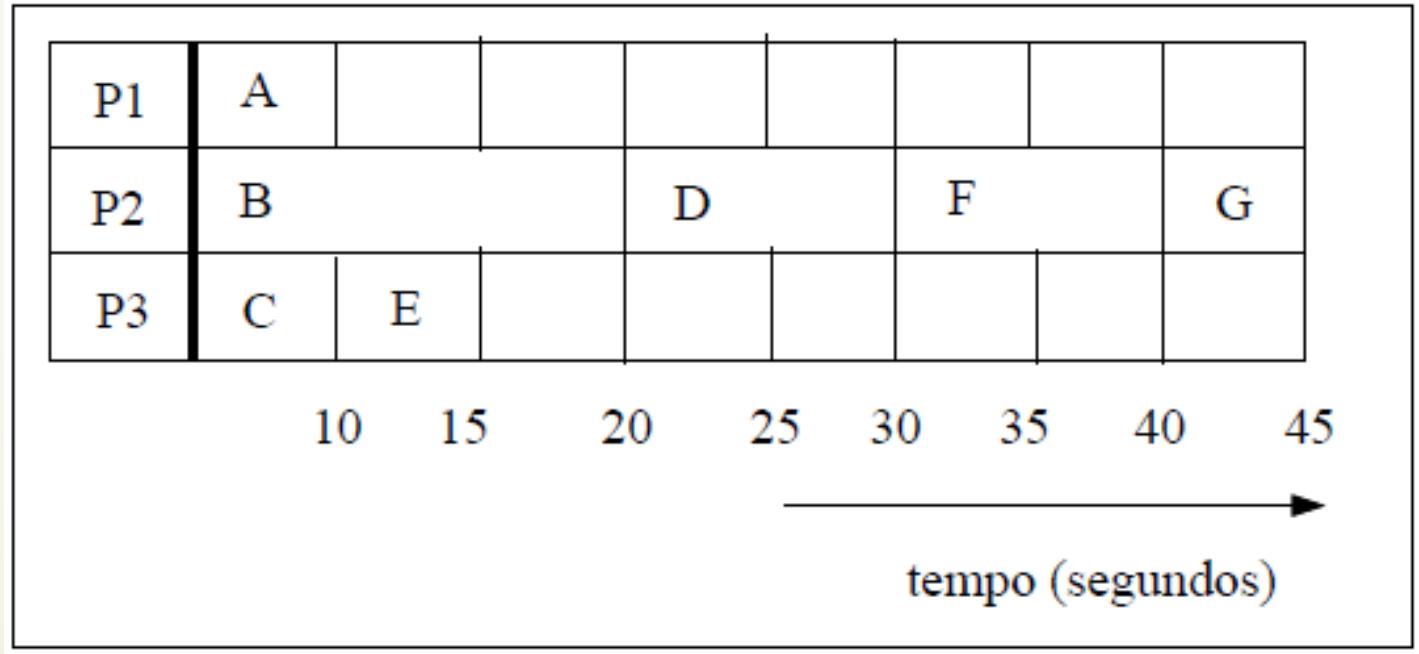
Paralelização Automática

- Considerando os tempos de execução de todas as tarefas igual a 10 segundos, têm-se:
- Tempo sequencial (tseq) = 70 segundos
- Tempo paralelo (tpar) = 40 segundos
- *Speedup* = $\frac{40}{70}$ = 0,57



Paralelização Automática

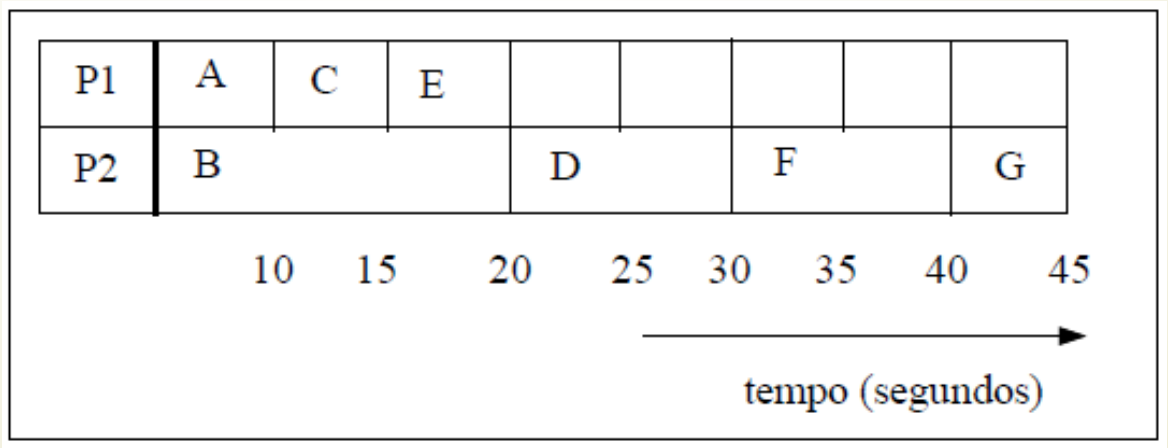
- Por outro lado, sejam 20,10 e 5 os tempos da exponenciação, multiplicação e adição respectivamente, então o diagrama seria:



- Nesse caso, o *speedup* seria de 45/70, ou 0,64.

Paralelização Automática

- Uma melhor configuração de tarefas, que geraria um melhor balanceamento e uma menor complexidade, seria a transferência das tarefas C e E para serem executadas em P1.



Paralelização Automática

- Seja o loop:

```
for i = 1 to n
    ai = 0
```

- Pode-se paralelizá-lo completamente, iniciando todas as i variáveis ao mesmo tempo:
  - a1 = 0 || a2 = 0 || a3 = 0 || ... || an = 0.

- Seja o seguinte bloco de comandos, dentro de uma instrução for:

```
for i=1 to n
    xi = 0;
    ai = bi + 1
```

- Pode-se paralelizar em primeiro nível e também podem-se executar as duas instruções em paralelo, tendo-se:
  - (x1 = 0 || x2 = 0 || ... || xn = 0) || (a1 = b1 + 1 || a2 = b2 + 1 || ... || an = bn + 1).

Paralelização Automática

- Seja o seguinte pedaço de código:

```
soma = 0;
for i = 1 to n
    soma = soma + ai
```

- Este trecho não é paralelizável, visto que a toda iteração da instrução *for* depende da iteração anterior.

Paralelização Automática X Paralelização Manual

- O desenvolvimento de algoritmos paralelos tem sido caracteristicamente manual.
- Tanto a identificação do paralelismo, como a sua implementação é responsabilidade do programador.
- O desenvolvimento pode ser bem demorado, é complexo e propenso a erros.

Paralelização Automática X Paralelização Manual

- Mas há um grande esforço por fazer essa paralelização de forma automática.
- O tipo mais comum de ferramenta para se paralelizar automaticamente um programa sequencial são os compiladores paralelizadores.

Paralelização Automática X Paralelização Manual

Totalmente automático:

- O compilador analisa o código-fonte e identifica oportunidades de paralelismo.
- A análise inclui uma ponderação do custo sobre se ou não o paralelismo realmente melhoraria o desempenho.
- Loops são o alvo mais frequente da paralelização totalmente automática.

Paralelização Automática X Paralelização Manual

Dirigido pelo Programador:

- Usando "diretivas do compilador" o programador diz explicitamente ao compilador como paralelizar o código.
- Pode ser capaz de ser usado em conjunto com um certo grau de paralelização automática também.
- Exemplo: Mecanismo de Threads disponível em linguagens de programação.

Paralelização Automática X Paralelização Manual

- O compilador mais comum para geração de algoritmos paralelos automaticamente é realizado em ambientes com memória compartilhada.
  - Exemplo: OpenMP

Paralelização Automática X Paralelização Manual

- Várias advertências importantes na paralelização automática:

Resultados errados podem ser produzidos

O desempenho pode degradar

Muito menos flexível do que a paralelização manual

Limitado a um subconjunto (laços em sua maioria) do código

Não se pode paralelizar código, se a análise do compilador sugere que o código é complexo demais

Conceitos Básicos

- Paralelismo Implícito
  - Quando é de responsabilidade do compilador e do sistema de execução:
    - Detectar o paralelismo do programa;
    - Atribuir as tarefas para execução paralela;
    - Controlar e sincronizar toda a execução.

Conceitos Básicos

- Paralelismo Explícito
  - Quando é responsabilidade do programador
    - Detectar o paralelismo do programa;
    - Atribuir as tarefas aos processadores;
    - Controlar a execução, indicando os pontos de sincronização;
    - Conhecer a arquitetura do computador, de forma a otimizar o desempenho

Conceitos Básicos

- Qual o número ideal de computadores para executar uma aplicação?
- Qual o número ideal de processadores para realizar uma tarefa?
- Qual a potência e a função de cada um?

Conceitos Básicos

- Níveis de Paralelismo (Granulosidade / Granulação)
  - Relaciona o tamanho das unidades de trabalho submetidas aos processadores.
  - Diversas definições de granulação podem ser encontradas na literatura
  - Esta é uma definição muito importante na computação paralela, visto que está intimamente ligada ao tipo de plataforma (o porte e a quantidade de processadores) à qual se aplica o paralelismo.

Conceitos Básicos

- Níveis de Paralelismo (Granulosidade / Granulação)
  - **Fina**: paralelismo de baixo nível
  - Nível de instruções ou operações
  - Grande número de processos, pequenos e simples.

Fácil de conseguir balanceamento de carga eficiente.

O tempo de computação nem sempre compensa os custos de criação, comunicação e sincronismo.

Difícil conseguir aumentar o desempenho.

Conceitos Básicos

- Níveis de Paralelismo (Granulosidade / Granulação)
  - **Grossa**: paralelismo de alto nível
  - Nível de programas
  - Geralmente observado em plataformas com poucos processadores, grandes e complexos.

O tempo de computação compensa os custos de criação, comunicação e sincronização.

Oportunidade de conseguir melhoras significativas no desempenho.

Difícil de conseguir balanceamento de carga eficiente.

Conceitos Básicos

- Níveis de Paralelismo (Granulosidade / Granulação)
  - **Média:** paralelismo de nível médio
  - Algo entre as duas anteriores
  - Dezenas de processos de tamanho médio

Programação Concorrente

- Para termos aplicações concorrentes são necessárias primitivas para a ativação e término de processos concorrentes.
- Diversas destas primitivas (ou comandos) tem sido propostas.
- Algumas delas são:
  - Fork / Join;
  - Cobegin / Coend;
  - Doall.

Programação Concorrente

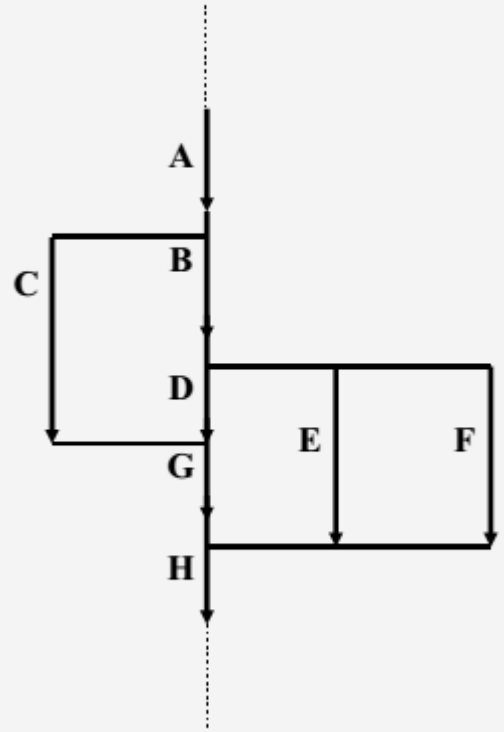
- Fork / Join
  - Ativa dois processos paralelos.
  - O comando *fork* inicia um novo processo (filho), concorrentemente ao processo que está sendo executado (pai).
  - O comando *join,* por sua vez, é utilizado para sincronização do pai com os filhos gerados.

Programação Concorrente

- Fork / Join
  - As sintaxes dos comandos *fork/join* são:
    - ***Fork* endereço**: executa o processo que está armazenado em endereço concorrentemente ao processo atual.
    - ***Join* num, end1, end2**: decrementa num em uma unidade e verifica seu valor. Se num for igual a zero, é executado o processo de end1, senão executa o processo de end2.

Programação Concorrente

```
    A
    m = 2
    fork c
    B
    n = 3
    fork e
    fork f
    D
    Join m, g, quit
g:  G
    join n, h, quit
h:  H
    quit
c:  C
    join m, g, quit
e:  E
    join n, h, quit
f:  F
    join n, h, quit
```



Programação Concorrente

- Fork / Join
  - A utilização de *fork/join* é um meio poderoso e flexível de se especificar o processamento concorrente.
  - Porém, programas escritos utilizando tais comandos devem ser escritos de maneira disciplinada, visto que a organização do código fonte obtido é desestruturada.
  - Com o objetivo de proporcionar maior estruturação, à custa de perda de flexibilidade, foram propostos alguns novos modelos.

Programação Concorrente

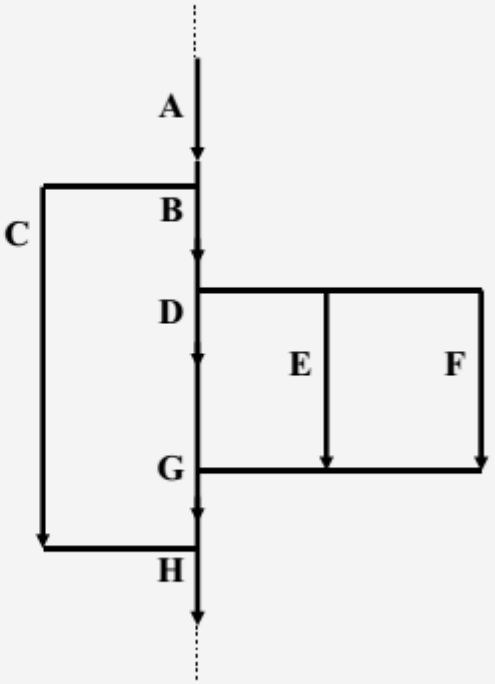
- Cobegin / Coend
  - Também chamados *parbegin/parend*, é uma forma mais estruturada para se ativar um conjunto de instruções que serão executadas em paralelo.
  - É mais alto nível que a construção *Fork/Join*, então fica fácil reconhecer quais processos estão executando em paralelo.

Programação Concorrente

- Cobegin / Coend
  - A execução concorrente das declarações S1, S2, ..., Sn pode ser ativada através da estrutura:
  - ***Cobegin S1 // S2 // S3 // ... // Sn Coend***
  - O processo pai será bloqueado até que S1, S2, ..., Sn estejam terminadas.

Programação Concorrente

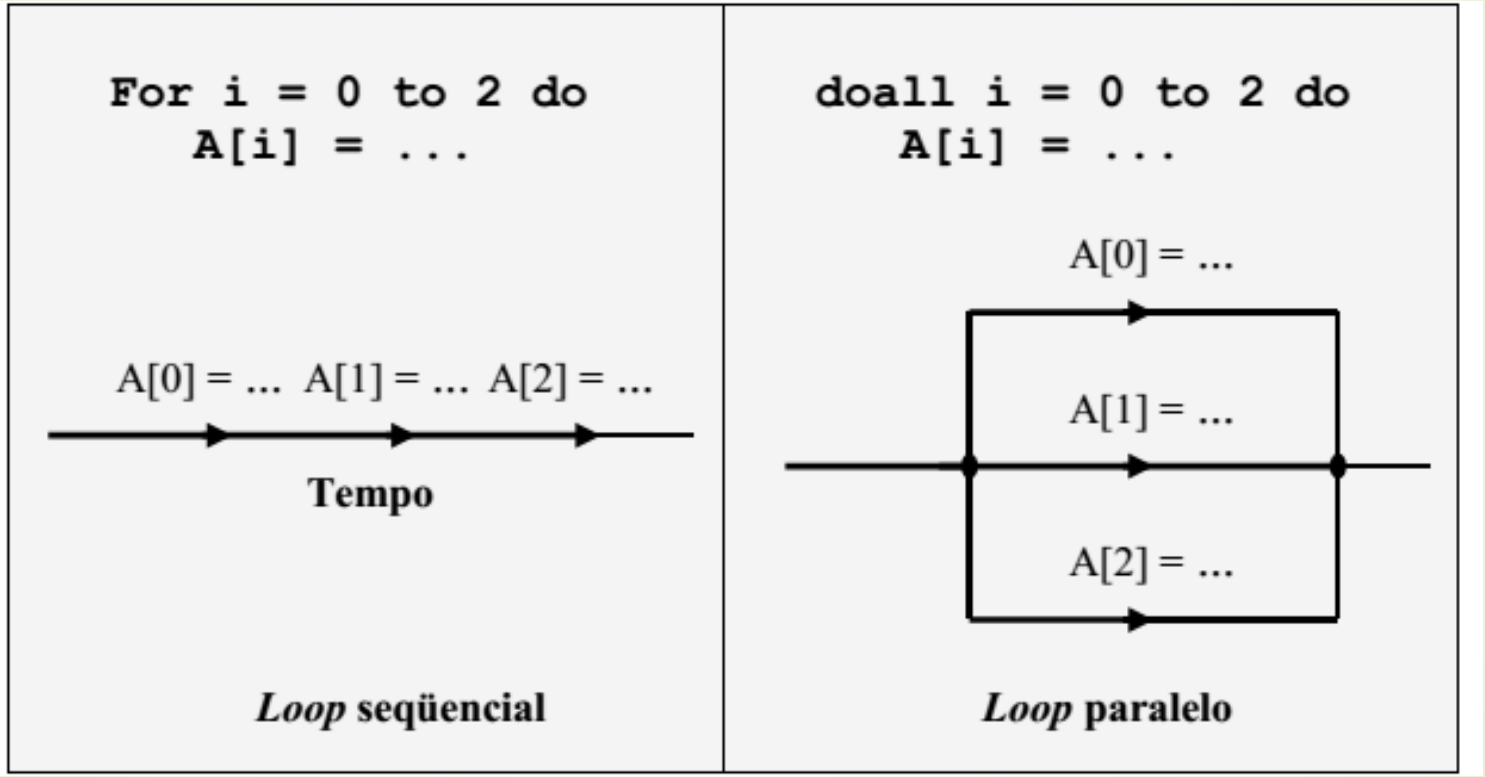
```
A
cobegin
    C
    begin
        B
        cobegin
            D
            E
            F
        coend
        G
    end
coend
H
```



Programação Concorrente

- Doall
  - Executa em paralelo as instâncias de um bloco de comando dentro de loops, desde que não haja dependência entre essas instâncias.
  - Muito utilizada para ser trabalhar com vetores e matrizes.

Programação Concorrente



Programação Concorrente

- Resumindo:
  - *Fork/join e cobegin/coend* são utilizados para a ativação de processos paralelos e *doall* para a ativação paralela de instâncias de loops.
  - *Em relação ao* contraste flexibilidade/estruturação, *fork/join* oferece um mecanismo flexível porém desestruturado, enquanto *cobegin/coend e doall* apresentam maior estruturação, o que diminui a flexibilidade.

Mapeamento de Processos

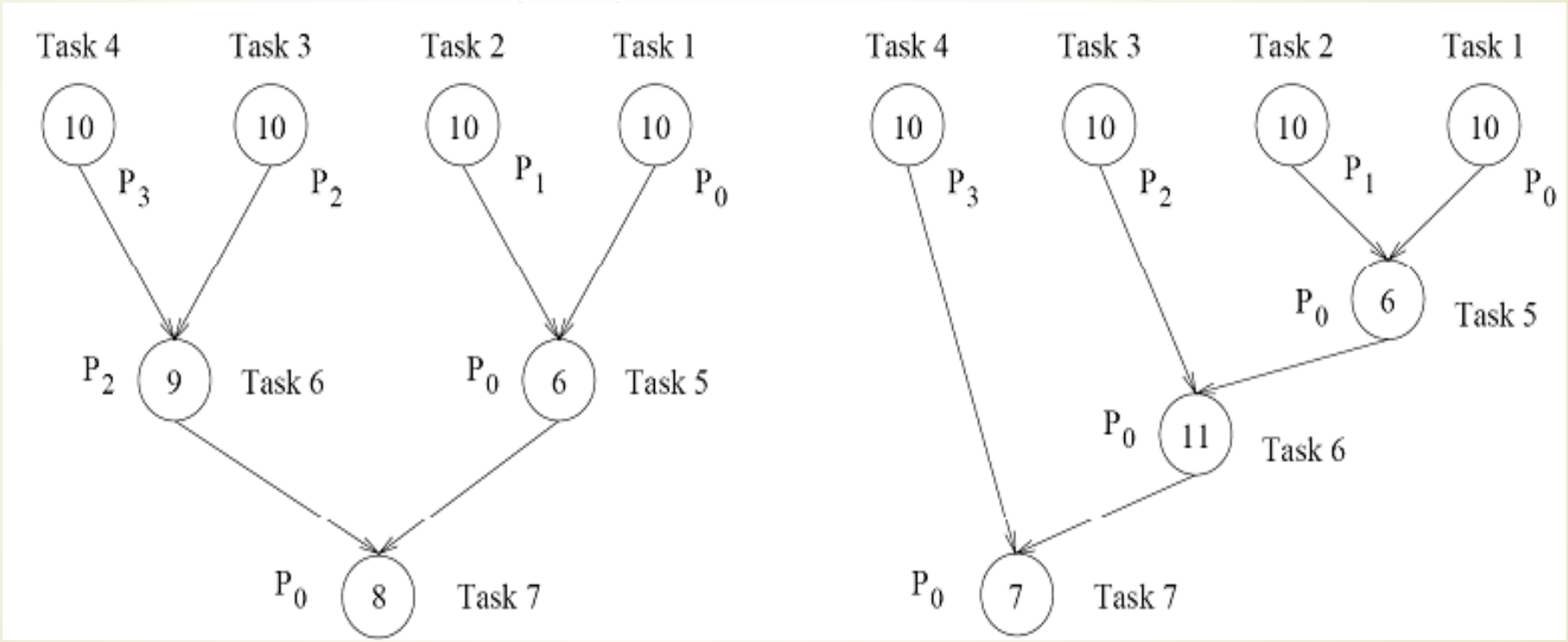
- Processo de atribuir tarefas aos processadores, de forma que a porcentagem de ocupação seja maximizada e a comunicação entre processos seja minimizada.
- A percentagem de ocupação é ótima se a computação é distribuída igualmente pelos processadores.
- Todos começam e terminam suas tarefas simultaneamente.
- A porcentagem de ocupação decresce quando processadores ficam ociosos.

Mapeamento de Processos

- Regras Básicas

1 - Atribuição de tarefas independentes a processos diferentes.
2 - Repartição do trabalho de forma equitativa entre todos os processos.
3 - A interação inter processos deve ser menor possível.
4 - Tarefas que apresentam interações elevadas devem ser executadas pelo mesmo processo.
5 - Atribuição das tarefas do caminho crítico aos processos assim que estejam reunidas as condições de execução (processo livre ou dados disponíveis)

Mapeamento de Processos



Mapeamento de Processos

O mapeamento mais eficaz das 2 decomposições da *query* anterior é obtido através das seguintes considerações:

- Utilização de 4 processos de forma a permitir o maior grau de paralelismo do algoritmo
- Minimização de interações obtida através da utilização do mesmo processo para tarefas dependentes de dois níveis sucessivos.

Técnicas de Decomposição

- Não existe um padrão para todos os problemas
- Existem algumas técnicas para decomposição de algoritmos em tarefas:
  - Decomposição recursiva
  - Decomposição de dados
  - Decomposição exploratória
  - Decomposição especulativa

Decomposição Recursiva

- Utilizada em algoritmos que possam ter uma solução do tipo dividir e conquistar.
- O problema inicial é decomposto em subproblemas de resolução independente, que por sua vez tem tratamento idêntico ao inicial.
- Essa decomposição é feita recursivamente até se atingir a granularidade pretendida.

Decomposição Recursiva - Exemplo

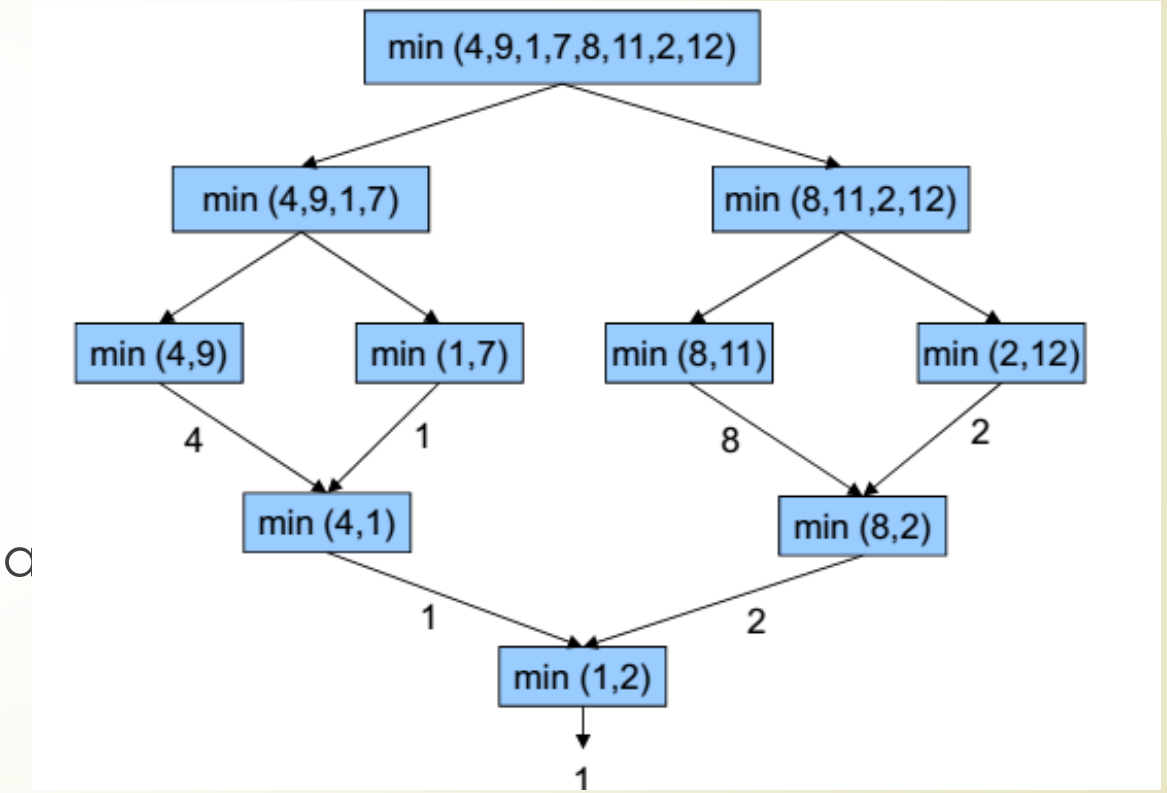
- O cálculo do menor valor de uma lista de números.

- Vamos considerar o seguinte caso:

{4, 9, 1, 7, 8, 11, 2, 12}

Decomposição Recursiva - Exemplo

- Um possível grafo de dependências:
- Podemos dividir o problema em dois subproblemas
- Cada um calcula o mínimo de metade dos elementos da lista
- E prosseguir recursivamente



Decomposição Baseada nos Dados

- A ideia é identificar a decomposição através dos dados que estão sendo computados.
  - Esses dados são divididos em subconjuntos.
- Cada tarefa será responsável por processar um subconjunto dos dados que lhe é atribuído.
- Muito utilizada no tratamento de grandes quantidades de dados.
- A forma como os dados são decompostos condiciona o desempenho da solução.

Decomposição Baseada nos Dados

- Pode-se levar em consideração:

Dados de Entrada

Dados de Saída

Dados Intermediários

Decomposição Baseada nos Dados

- Dados de saída:
  - Em alguns casos, os dados de saída são obtidos exclusivamente em função dos dados de entrada.
  - A decomposição é realizada através da atribuição de uma tarefa para cálculo sob cada subconjunto dos dados.

Decomposição Baseada nos Dados

- Dados de saída
- Exemplo: Multiplicação de Matrizes:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Decomposição Baseada nos Dados

- Pode-se ter mais de uma decomposição baseada nos dados de saída.

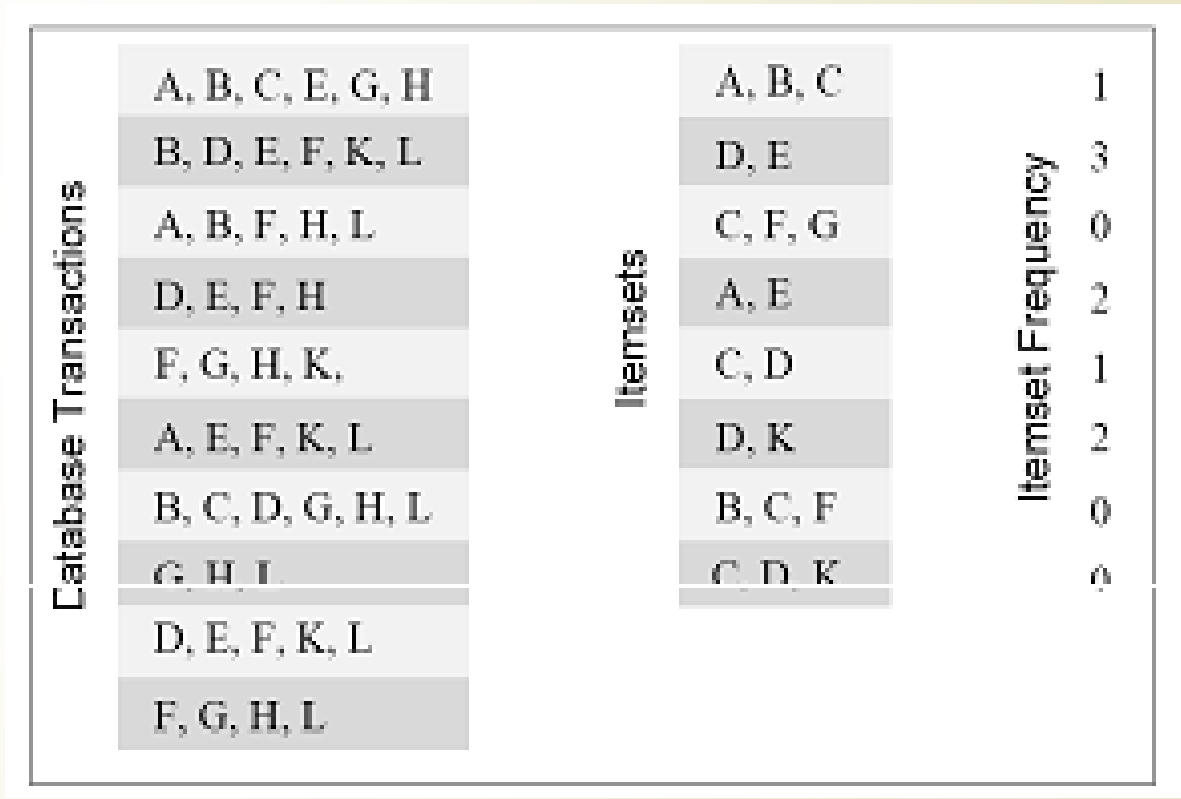
Decomposition I	Decomposition II
Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

Decomposição Baseada nos Dados

- Dados de Entrada:
  - Aqui, não é possível determinar previamente os dados de saída.
    - Ex. Ordenações e contagem de valores.
  - Mas é possível decompor os dados de entrada e atribuir uma tarefa a cada subconjunto os dados.

Decomposição Baseada nos Dados

- Dados de Entrada
- Exemplo: contar referências de certos itens em uma base de dados que registra transações.



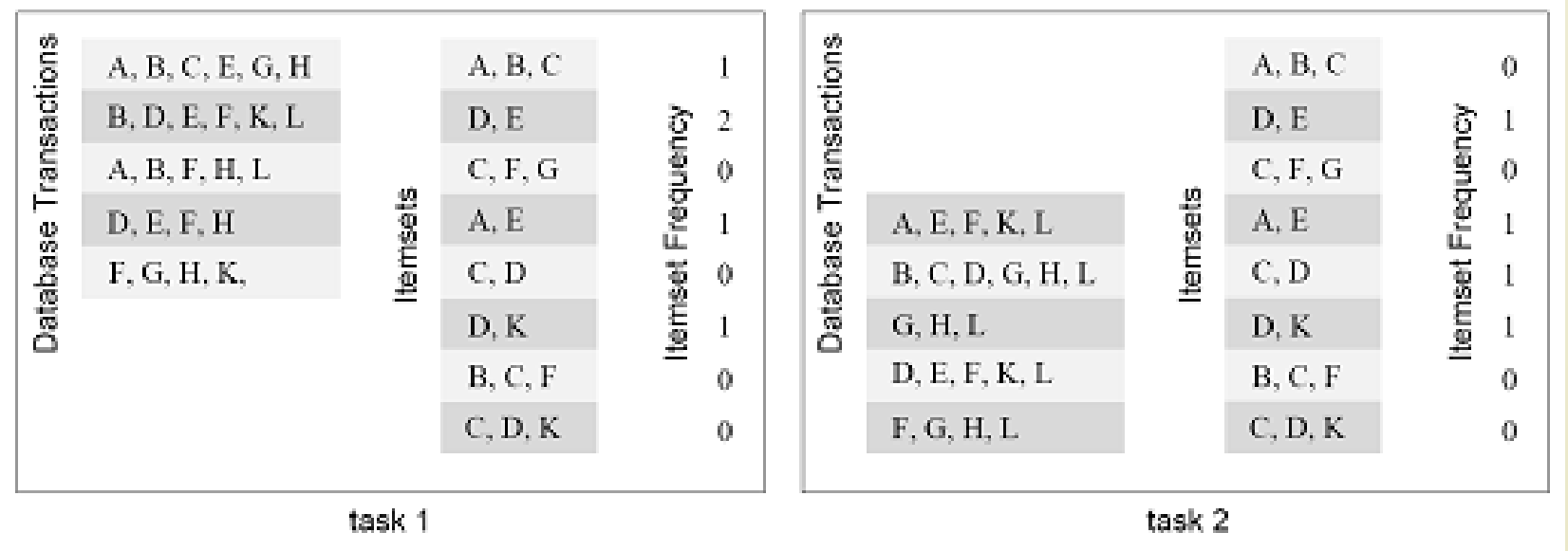
Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,
A, E, F, K, L
B, C, D, G, H, L
G, H, I
D, E, F, K, L
F, G, H, L

Itemsets
A, B, C
D, E
C, F, G
A, E
C, D
D, K
B, C, F
C, D, K

Itemset Frequency
1
3
0
2
1
2
0
0

Decomposição Baseada nos Dados

- Tanto o registro de transações quanto os itens a serem contados são as entradas do problema.
- Podemos dividir o conjunto de transações em dois subconjuntos e contar as ocorrências em cada um deles em duas tarefas distintas.

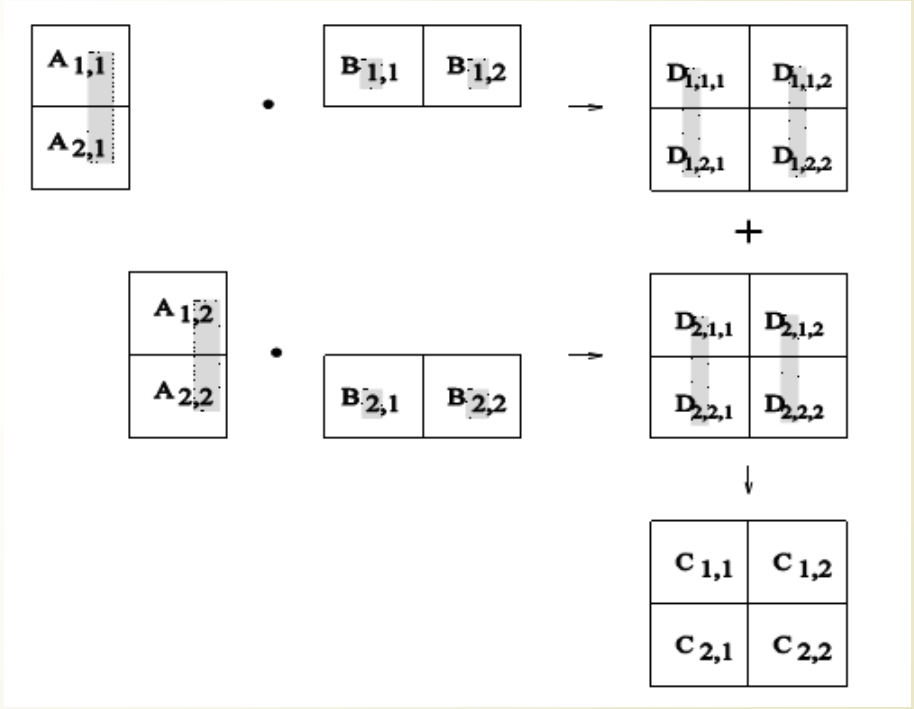


Decomposição Baseada nos Dados

- Dados Intermediários:
  - Em outros casos, uma computação pode ser encarada como uma sequência de transformações sucessivas.
  - E a decomposição também pode ser feita em passos intermediários.

Decomposição Baseada nos Dados

- O produto de C de duas matrizes A e B pode ser considerada a soma de duas matrizes D intermediárias obtidas da multiplicação de partes da matriz inicial.
- Nesse caso, a decomposição é feita em função dos dados das saídas das matrizes intermediárias.



Decomposição Baseada nos Dados

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \left(\begin{Bmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{Bmatrix} \begin{Bmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{Bmatrix} \right)$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Decomposição Baseada nos Dados

Task 01: $D_{1,1,1} = A_{1,1} B_{1,1}$

Task 02: $D_{2,1,1} = A_{1,2} B_{2,1}$

Task 03: $D_{1,1,2} = A_{1,1} B_{1,2}$

Task 04: $D_{2,1,2} = A_{1,2} B_{2,2}$

Task 05: $D_{1,2,1} = A_{2,1} B_{1,1}$

Task 06: $D_{2,2,1} = A_{2,2} B_{2,1}$

Task 07: $D_{1,2,2} = A_{2,1} B_{1,2}$

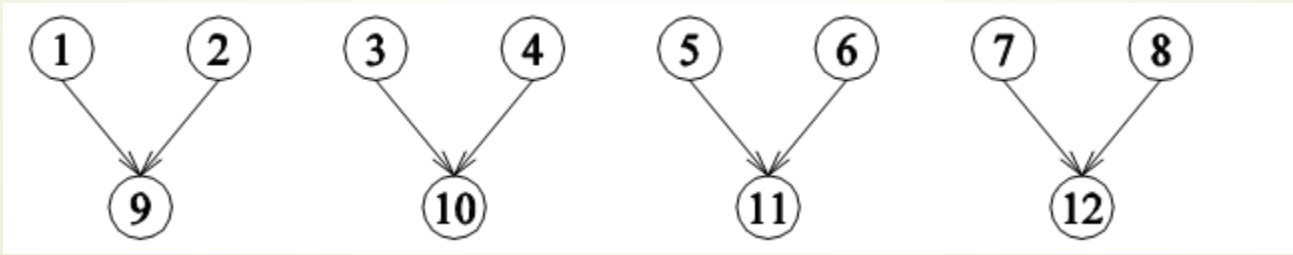
Task 08: $D_{2,2,2} = A_{2,2} B_{2,2}$

Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$

Task 12: $C_{2,,2} = D_{1,2,2} + D_{2,2,2}$



Decomposição Baseada nos Dados

- Dados Intermediários:
  - Esta abordagem pode trazer resultados melhores em cenários com matrizes muito grandes, em que uma decomposição baseada na saída não introduz o grau de paralelismo adequado à plataforma.
    - Ex. processadores vetoriais.

Decomposição Baseada nos Dados

- As decomposições baseadas em entrada ou saída geralmente são designadas pela regra "o dono é que trata".
  - O processo ao qual é atribuído um subconjunto de dados é responsável por toda a computação associada.
  - Torna simples o mapeamento das tarefas.
  - Pode ser definida de forma estática (tempo de projeto) ou dinâmica (tempo de execução).

Decomposição Exploratória

- Aplicada a problemas em que a decomposição está associada as fases de execução do algoritmo, e não ao conjunto de dados.
- Problemas que envolvem a exploração de espaços de possíveis soluções.

Análise combinatória

Problemas de otimização discreta

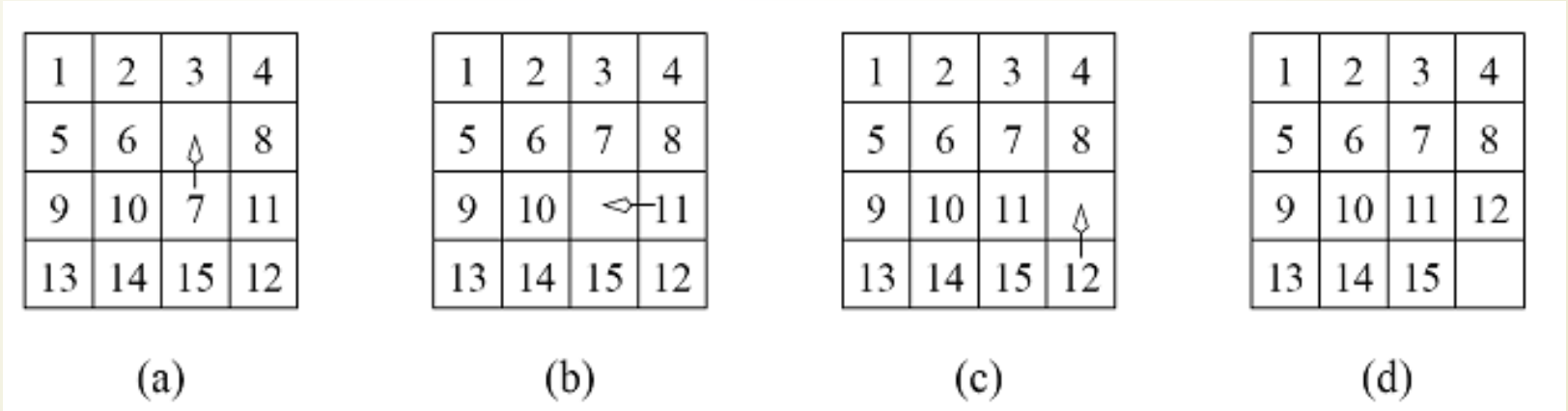
Prova de teoremas

Jogos

Criptoanálise

Decomposição Exploratória

- Exemplo: descobrir o conjunto de jogadas que permite chegar a solução de um puzzle de 15 peças móveis.

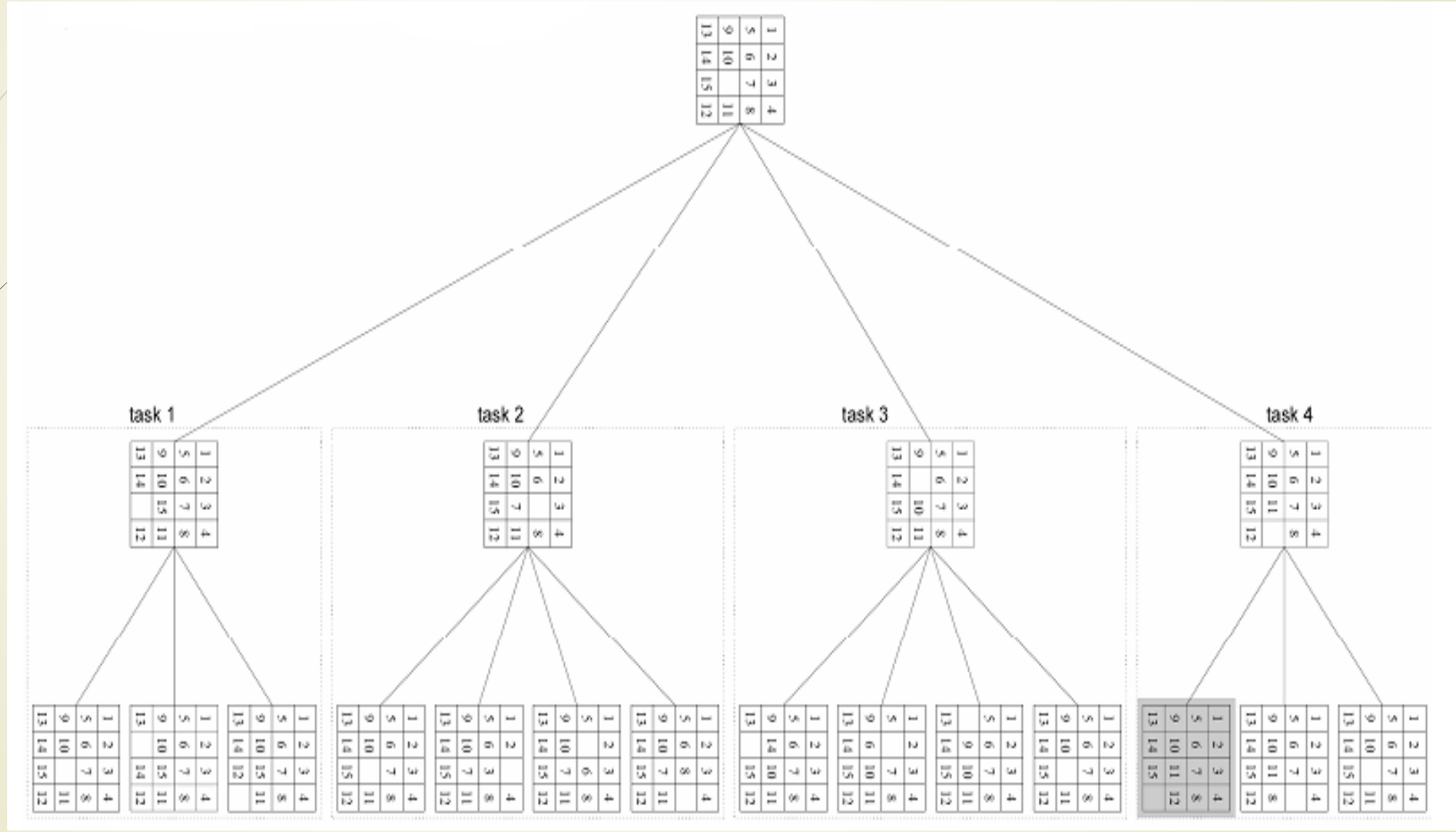


- A sequência A, B, C, D leva à solução do problema.

Decomposição Exploratória

- Exemplo: descobrir o conjunto de jogadas que permite chegar a solução de um puzzle de 15 peças móveis.
  - A decomposição paralela é feita gerando um primeiro nível de soluções possíveis e atribuindo cada solução a uma tarefa que realiza a exploração do respectivo espaço.
  - Quando uma solução intermediária é encontrada por uma das tarefas, as que não tiveram êxito devem ser interrompidas e atribuídas à exploração de novos espaços gerados a partir do novo estado.

Decomposição Exploratória



Decomposição Especulativa

- Em outras aplicações, as dependências entre tarefas não são conhecidas à priori.
  - Dependem de resultados intermediários.
- Impossível identificar tarefas dependentes.

Decomposição Especulativa

- Podem ser realizadas duas aproximações.

Conservadora

- Só considera tarefas independentes quando há garantia de não existir dependências.
- Pode-se obter níveis de paralelismo reduzidos.

Decomposição Especulativa

- Podem ser realizadas duas aproximações.

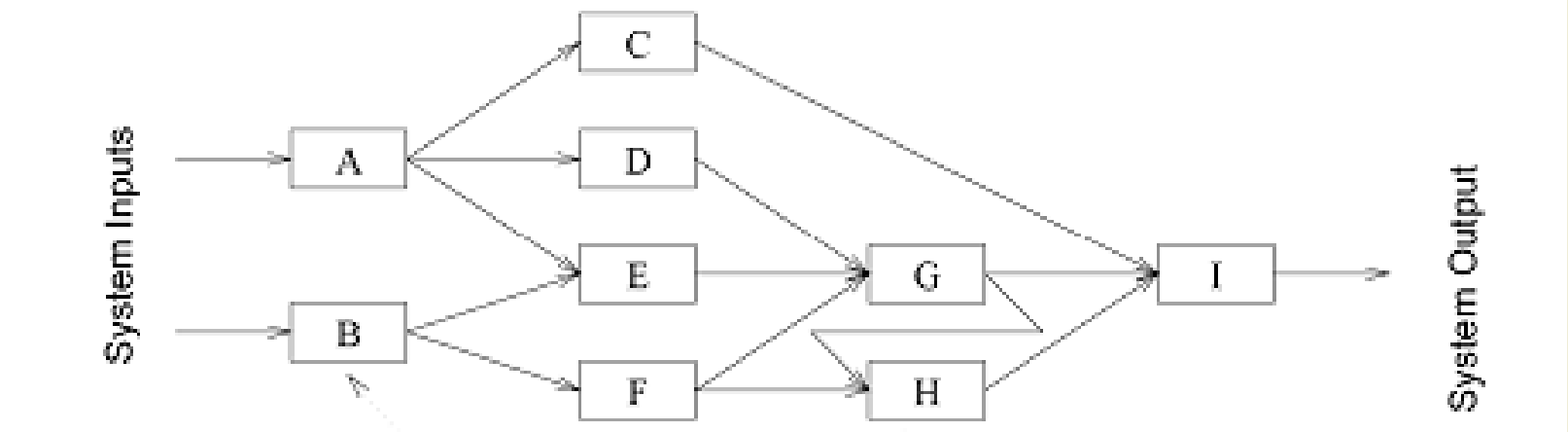
Otimista

- Cria tarefas independentes mesmo quando não existe a garantia, confiando em mecanismos de avaliação posteriori das dependências.
- Necessário mecanismos de controle e rollback, para desfazer algum trabalho pela antecipação.

Decomposição Especulativa - Exemplo

- Seja o caso de uma aplicação de simulação de um sistema por vários vértices de execução ligados por um conjunto de arestas direcionadas.
  - Uma rede de computadores ou uma linha de montagem de uma fábrica
- O valor da saída de um vértice pode condicionar o restante do caminho.

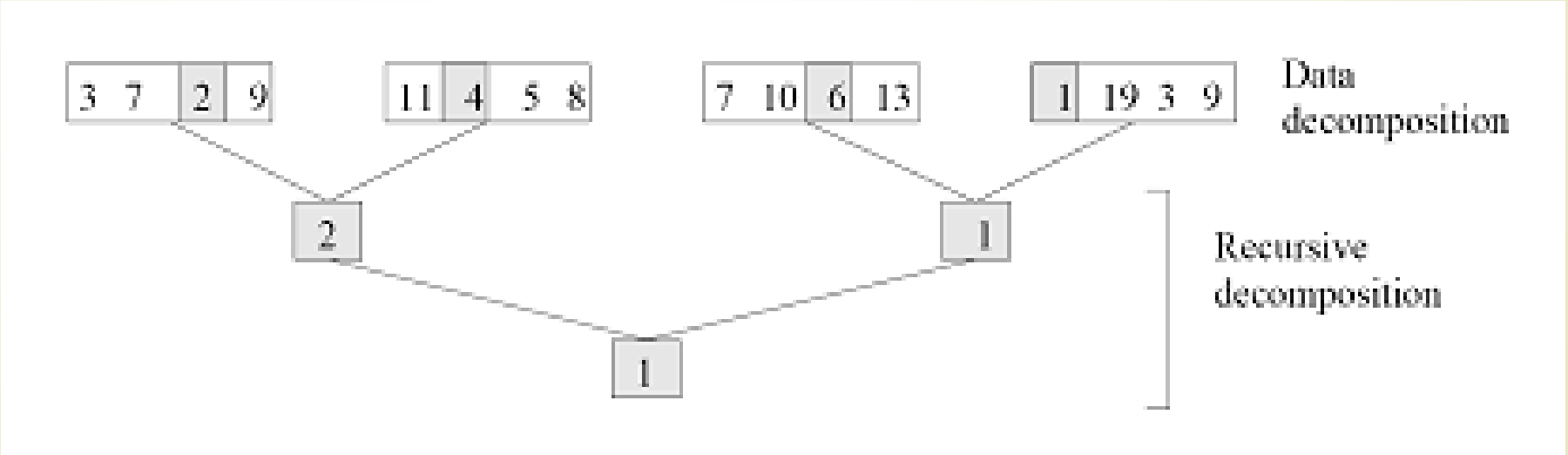
Decomposição Especulativa - Exemplo



É possível antecipar por exemplo o processamento do vértice E (probabilidade de escolha é maior) com valores simulados antes de se saber se será efetivamente a escolha certa

Decomposição Híbrida

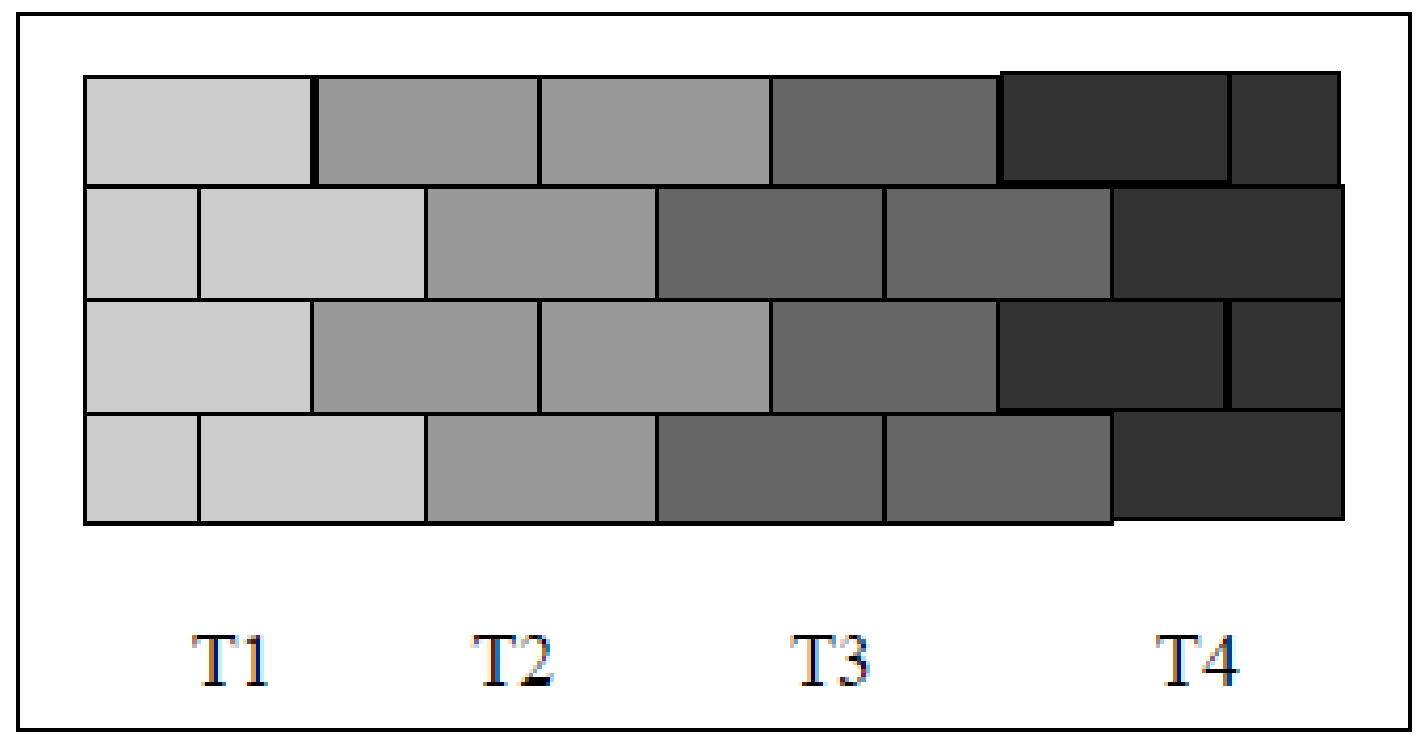
- Na maioria dos casos, é utilizada mais de uma dessas técnicas para decompor eficazmente um problema.
- Se considerarmos como exemplo de encontrar o mínimo de uma lista de números, poderíamos utilizar a decomposição de dados e também a recursiva



Paralelismo Geométrico

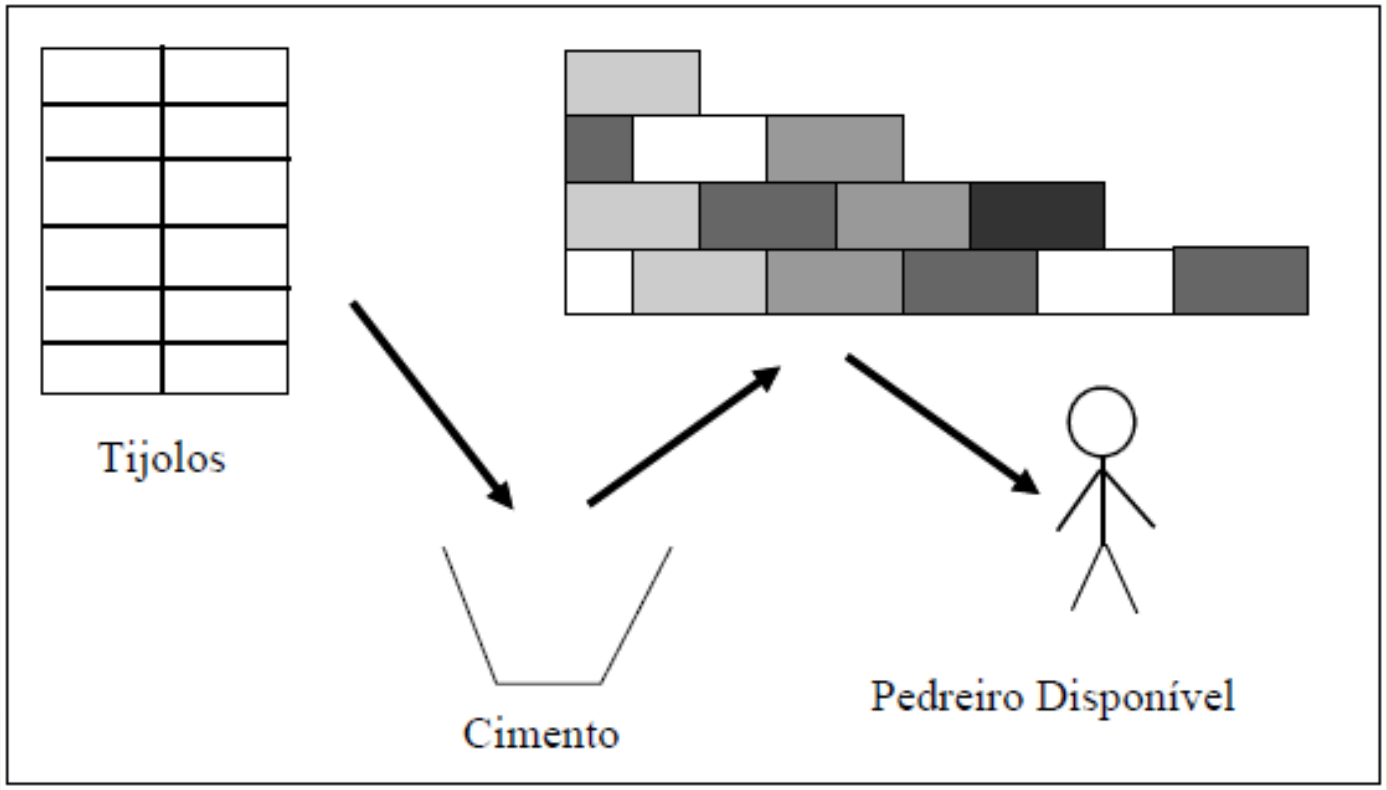
- Divisão do conjunto de dados a serem trabalhados igualmente entre todos os processadores
- Cada processador executa uma cópia do programa completo, porém em um subconjunto de dados
- Modo mais fácil de desenvolvimento de algoritmos paralelos

Paralelismo Geométrico



Paralelismo “Processor Farm”

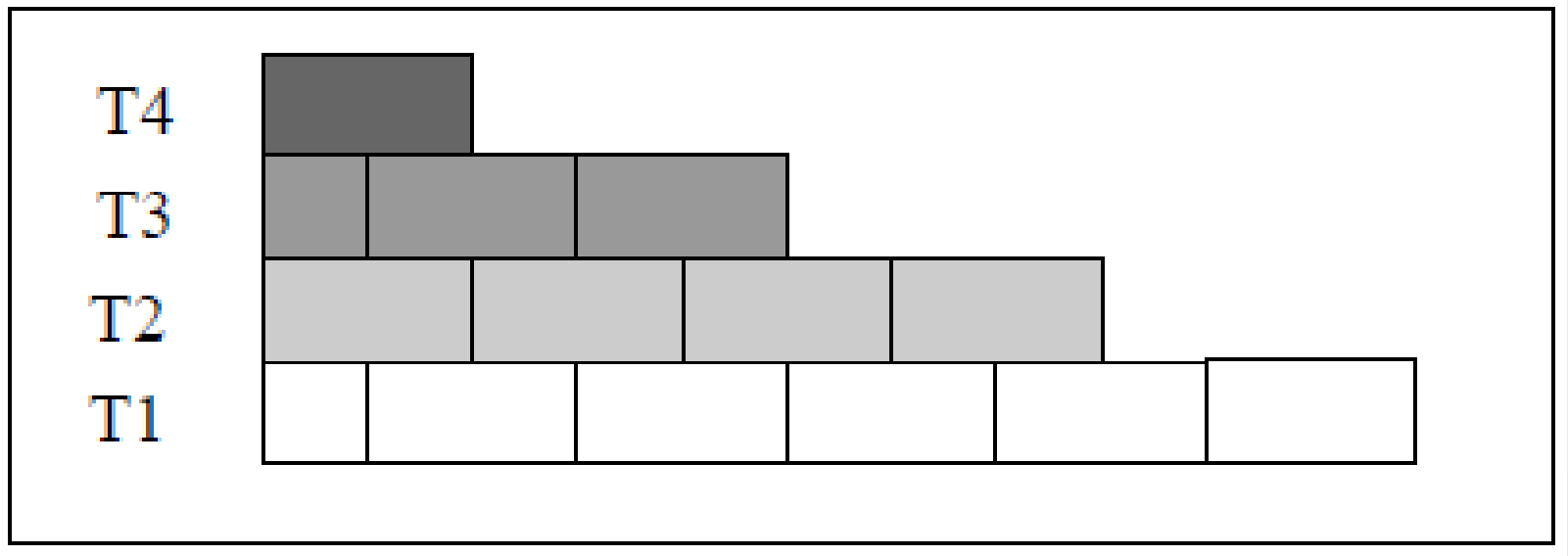
- Caracteriza-se pela existência de um processador “mestre” que supervisiona um grupo de processadores “escravos”



Paralelismo “Processor Farm”

- Vantagens
  - Facilidade de ampliação do sistema, o que pode ser conseguido através do aumento de trabalhadores
  - facilidade de programação
  - balanceamento de carga mais natural, visto que as tarefas vão sendo submetidas aos processadores de acordo com a disponibilidade
- Desvantagens
  - a sobrecarga de comunicação
  - a possibilidade de gargalo no processador “mestre”

Paralelismo *Pipeline*



Grau de Concorrência

- O número de tarefas que podem ser executadas em paralelo determina o grau de concorrência de uma decomposição.
- O grau de concorrência aumenta quando a decomposição se torna mais fina, e vice-versa.

Grau de Concorrência

- O grau de concorrência varia ao longo da execução de uma tarefa. Temos então:

Grau de concorrência máximo:

Máximo de tarefas que podem ser executadas em paralelo num dado instante.

Grau de concorrência médio:

Média de tarefas que podem ser executadas em paralelo ao longo da execução do algoritmo.

Grau de Concorrência

- O grau de concorrência médio é determinado relacionando o valor total do custo de execução e o comprimento do caminho crítico.

- Caminho Crítico: o caminho mais longo no grafo de dependências.

Grau de Concorrência

- A soma dos pesos de todos os vértices ao longo desse caminho é dado por:

$$C_P = \sum w_i$$

(i: vértices do caminho crítico)

Grau de Concorrência

- O custo total de execução é a soma de todos os vértices do grafo de dependências.

$$W_t = \sum_i$$

(i: todos os vértices)

Grau de Concorrência

- O grau de concorrência médio é determinado através do quociente entre o valor do custo de execução total e o custo de execução do caminho crítico, e mede a eficácia de uma decomposição.
- O objetivo é sempre maximizar o grau de concorrência!

$$A_V = W_t / C_p$$

Grau de Concorrência

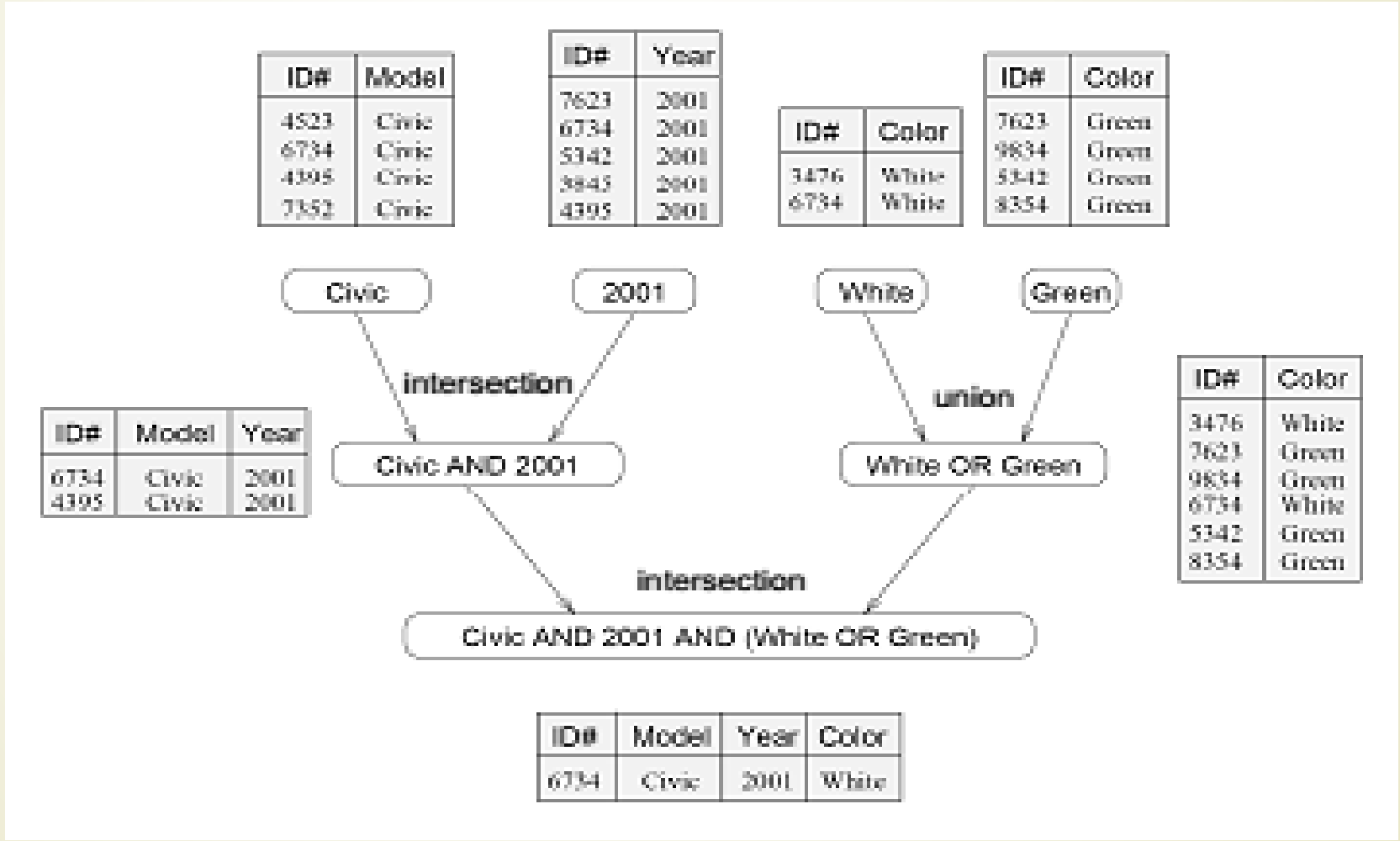
- Exemplo: processamento de *queries* em um BD.
- A execução da query pode se decomposta em sub tarefas de várias maneiras
- Cada uma gerará uma tabela intermediária que resulta da aplicação de uma cláusula.

MODEL = ``CIVIC'' AND YEAR = 2001 AND
(COLOR = ``GREEN'' OR COLOR = ``WHITE)

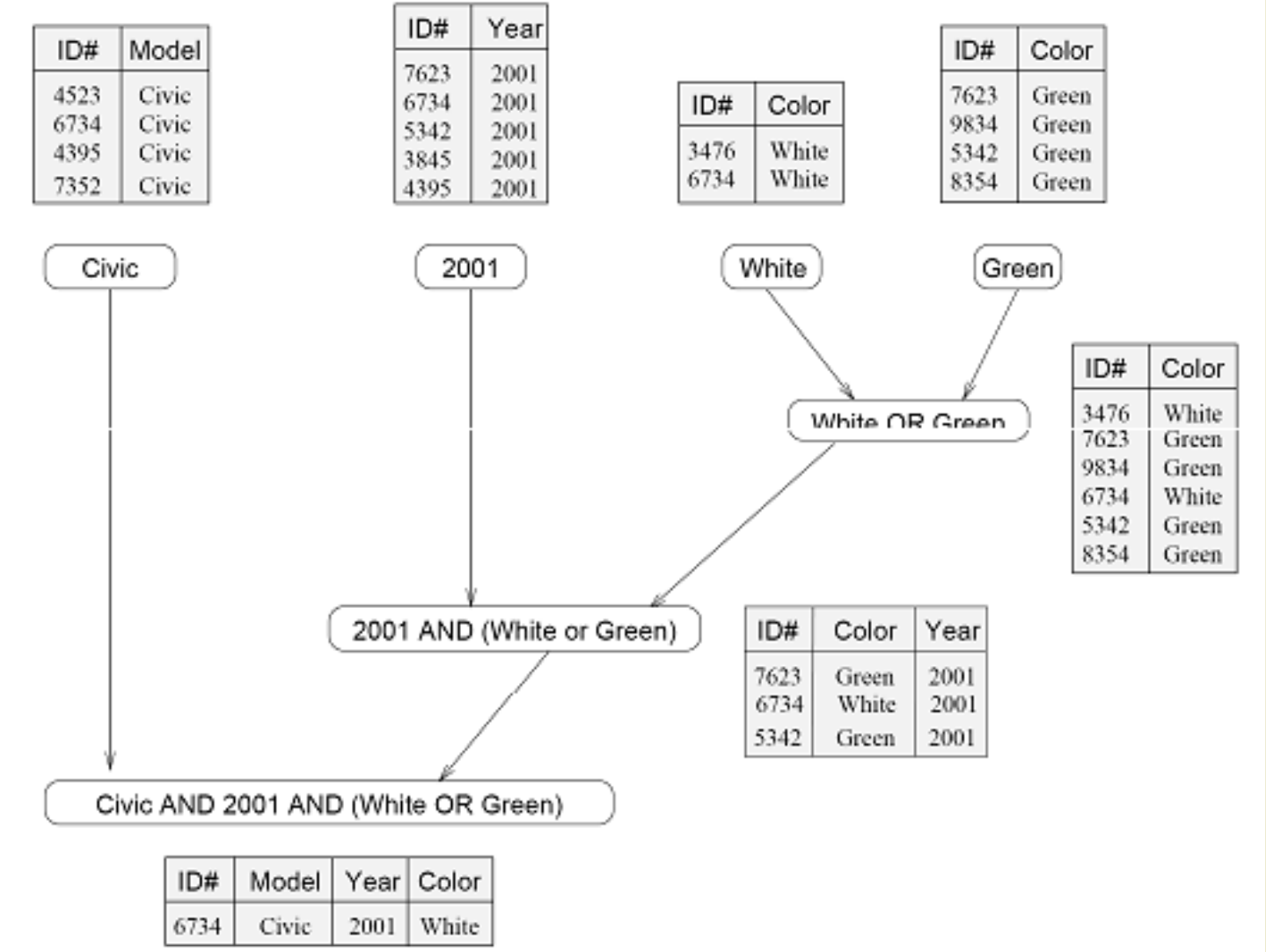
Na tabela de base de dados seguinte:

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Grau de Concorrência



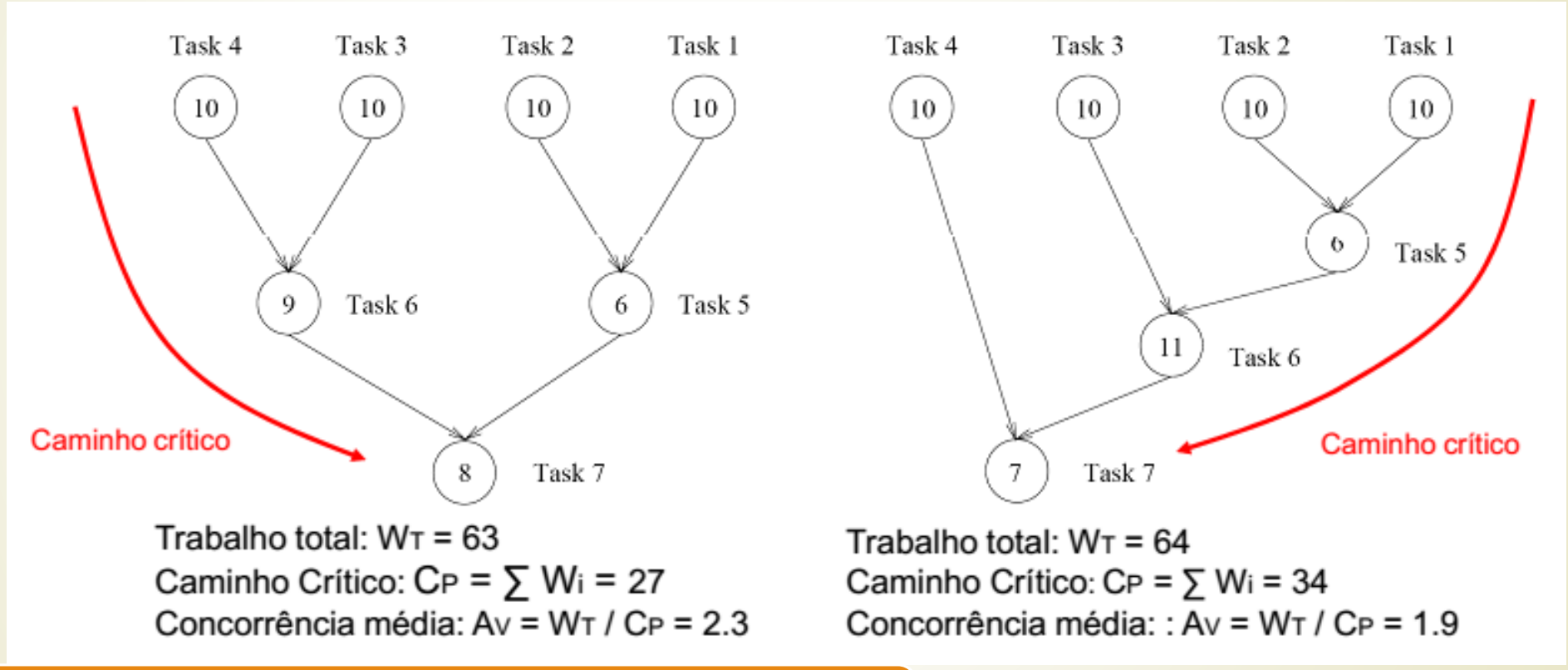
Grau de Concorrência



Grau de Concorrência

- Se considerarmos os dois grafos de dependências das duas situações... (o número nos vértices representa de forma simplificada os custos de execução de cada tarefa)
- Temos:

Grau de Concorrência



Qual decomposição é mais eficaz?

A primeira decomposição é mais eficaz que a segunda.

Mecanismos de Sincronização e Comunicação

- Em aplicações paralelas, os processos precisam comunicar entre si.
- Esta comunicação pode ser implementada de diversas maneiras:
  - Variáveis compartilhadas
  - Memória principal
  - Troca de mensagens

Mecanismos de Sincronização e Comunicação

- Comunicação é necessária para que processos interagindo na resolução de determinada aplicação troquem informações.
- E quando há comunicação, devem existir operações de sincronização, para fornecer controle de acesso e controle de sequência.

Mecanismos de Sincronização e Comunicação

- Controle de sequência (também chamado de sincronização condicional ou sincronização de atividades)
  - Utilizado para que se determine uma ordem na qual os processos (ou partes deles) devem ser executados.
- Controle de acesso
  - Necessário quando há competição entre processos para a manipulação de algum recurso.
  - Deve-se garantir que acessos concorrentes a esses recursos sejam controlados, para que se mantenha a consistência

Mecanismos de Sincronização e Comunicação

Resumindo: podemos dizer que a comunicação permite que a execução de um processo influencie na execução de outro.

E a sincronização impõe uma ordem na execução desses processos.

Mecanismos de Sincronização e Comunicação

- As técnicas que garantem a comunicação e o acesso a recursos compartilhados são conhecidos como mecanismos de sincronização.

- E é de fundamental importância que tais recursos sejam implementados para se garantir a integridade e confiabilidade das aplicações concorrentes.

Mecanismos de Sincronização e Comunicação

- Condição de disputa: alguns processos acessam simultaneamente uma mesma região, conhecida como região crítica.
- Os mecanismos de sincronização devem evitar a concorrência nas regiões críticas, de modo que em um determinando instante apenas um processo tenha acesso ao recurso.
- Essa ideia de exclusividade é denominada exclusão mútua.

Mecanismos de Sincronização e Comunicação

- 4 condições de exclusão mútua:

1. Dois ou mais processos não podem estar simultaneamente na mesma região crítica.
2. Nenhum processo fora da região crítica pode bloquear a execução de outro processo.
3. Nenhum processo deve esperar infinitamente (*starvation*) para ter acesso a uma região crítica.
4. A solução não deve fazer considerações sobre o número de processadores nem das velocidades dos mesmos.

Mecanismos de Sincronização e Comunicação

- A comunicação e o sincronismo pode ser necessária tanto em ambientes de memória compartilhada quanto em ambiente com memória distribuída.
- No caso da memória ser compartilhada, vários métodos podem ser utilizados.
  - Algumas técnicas são implementadas em hardware e outras em software.

Mecanismos de Sincronização e Comunicação – Memória Compartilhada

- Soluções em Hardware
  - Desabilitação de instruções: solução simples, que consiste em desabilitar as interrupções externas antes do processo entrar na região crítica, e as reabilite quando sair dessa região.
  - Produz uma série de inconvenientes, sendo que o maior deles acontece quando o processo que desabilitou as interrupções não volta a habilitá-las.

Mecanismos de Sincronização e Comunicação – Memória Compartilhada

- Soluções em Hardware
  - Instrução Test-and-Set: disponível em muitos processadores, permite ler uma variável, armazenar seu conteúdo em outro local da memória e atribuir um novo valor a essa variável.
  - Assim dois processos não podem manipular uma variável compartilhada ao mesmo tempo (exclusão mútua).

Mecanismos de Sincronização e Comunicação – Memória Compartilhada

- Soluções em Software
  - As primeiras soluções para a sincronização em software possuíam deficiências de espera ocupada.
  - O processo que não consegue acessar a região crítica, entra em looping até que consiga o acesso.
  - O processo consome tempo do processador desnecessariamente.
  - Outras soluções surgiram, e duas delas são as mais utilizadas:
    - Semáforos; e
    - Monitores.

Mecanismos de Sincronização e Comunicação – Memória Compartilhada

- Soluções em Software
  - Semáforos: uma variável inteira, não negativa, que só pode ser manipulada por duas instruções atômicas: down e up.
  - Estas instruções funcionam como protocolos de entrada e saída de uma região crítica.

Mecanismos de Sincronização e Comunicação – Memória Compartilhada

- Soluções em Software
  - Semáforos

```
Procedimento DOWN (S : Semáforo)
inicio
    se S = 0
       então Bloqueia_Processo
       senão S ← S - 1
    fim_se
fim


Procedimento UP (S : Semáforo)
inicio
    se Tem_Processo_Bloqueado
       então Libera_Um_Processo
       senão S ← S + 1
    fim_se
fim
```

Mecanismos de Sincronização e Comunicação – Memória Compartilhada

- Soluções em Software
  - Monitores: apresentam mais alto nível e oferecem uma alternativa mais estruturada para a implementação da exclusão mútua.
  - Um monitor é um conjunto de procedimentos, variáveis e estruturas de dados definidos em um módulo.

Mecanismos de Sincronização e Comunicação – Memória Compartilhada

- Soluções em Software
  - Monitores:
    - Toda vez que um processo faz uma chamada a um dos procedimentos, o monitor verifica se já existe outro processo executando algum procedimento do próprio monitor.
    - Se existir, o processo aguarda sua vez.
    - A implementação da exclusão mútua é realizada pelo compilador e não pelo programador.

Mecanismos de Sincronização e Comunicação – Memória Distribuída

- No caso de ambientes com memória distribuída, a comunicação e sincronização devem ser implementadas através da troca de mensagens.
- Realizada através das primitivas de SEND e RECEIVE.
- A rotina Send é responsável por enviar as mensagens para um processo receptor, e a Receive por receber uma mensagem de um processo transmissor.

Mecanismos de Sincronização e Comunicação – Memória Distribuída

- Alguns problemas também surgem na comunicação por troca de mensagens.
  - Por exemplo, pode ocorrer a perda da mensagem.
  - Para evitar este incidente, o processo receptor deve enviar ao transmissor uma confirmação do recebimento (ACK). E se o transmissor não receber um ACK em um determinado tempo, ele deve reenviar a mensagem.

Mecanismos de Sincronização e Comunicação – Memória Distribuída

Comunicação Síncrona

O processo que transmite a mensagem é bloqueado até que receba uma confirmação de recebimento da mensagem.

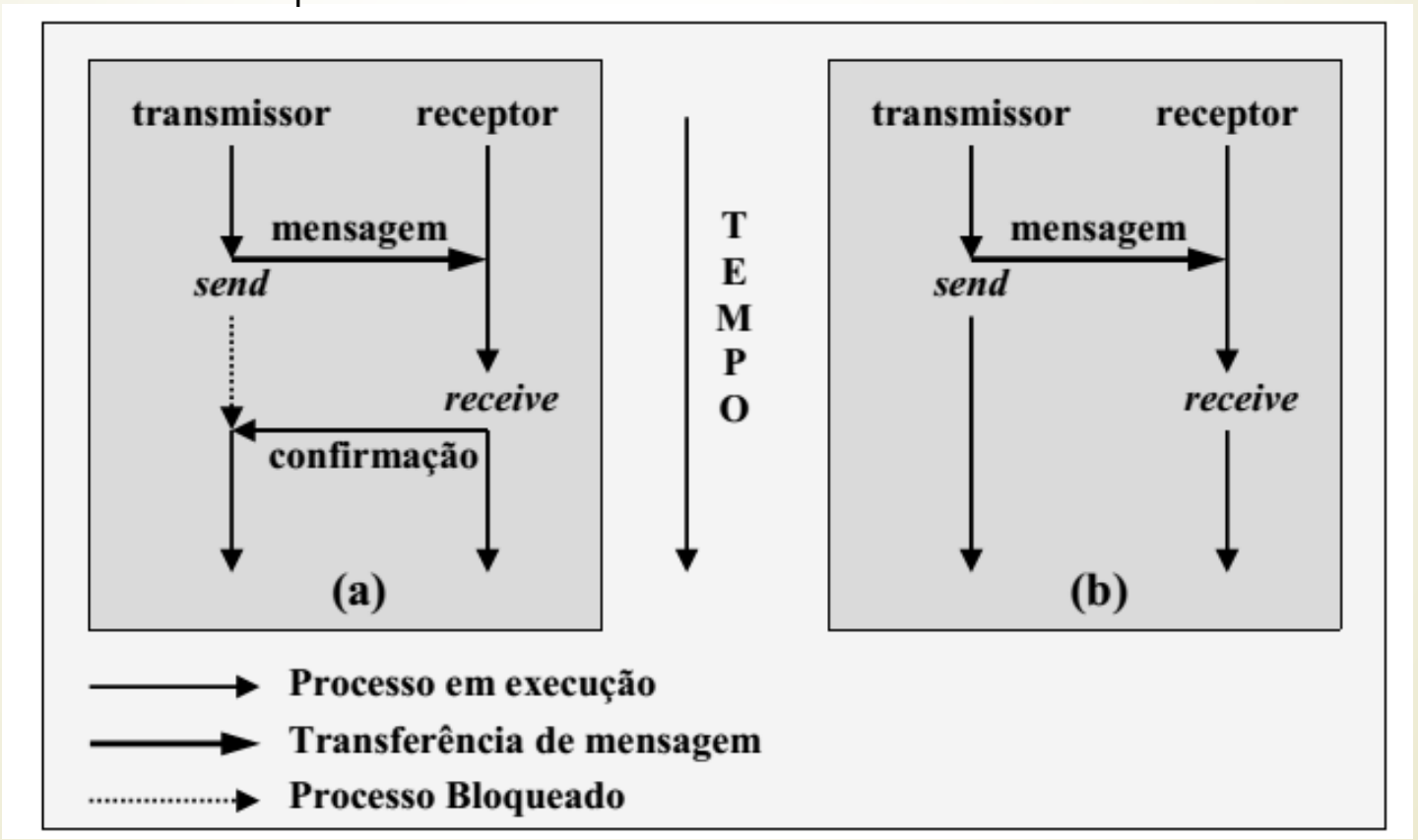
Comunicação Assíncrona

O processo transmissor envia a mensagem (que deve ser armazenada em um *buffer*) e continua sua execução.

Mecanismos de Sincronização e Comunicação – Memória Distribuída

Síncrona ou Bloqueante

Assíncrona ou Não Bloqueante



Projeto de um Algoritmo Paralelo

- 3 maneiras de construir um algoritmo paralelo:
  - Detectar e explorar algum paralelismo inerente a um algoritmo sequencial existente
    - abordagem muito utilizada
    - apresenta baixo speedup
    - não há necessidade de nova análise do algoritmo
  - Criar um algoritmo paralelo novo
    - possibilita melhor desempenho
    - necessita de reestruturação completa do algoritmo
  - Adaptar outro algoritmo paralelo que resolva problema similar
    - bom desempenho
    - exige menor trabalho do programador em relação à construção completa do algoritmo

Projeto de um Algoritmo Paralelo

- 3 aspectos importantes
  - Escolha da abordagem a ser seguida: tempo de escrita do algoritmo e o desempenho obtido
  - Pesar o custo da comunicação entre processos
    - operações de comunicação geram sobrecarga
  - Arquitetura do hardware paralelo

Projeto de um Algoritmo Paralelo

- Processo de desenvolvimento de algoritmos paralelos em 4 etapas:
  - Identificação do paralelismo inerente ao problema
  - Organização do trabalho
  - Desenvolvimento do algoritmo
  - Implementação

Projeto de um Algoritmo Paralelo

- Durante o desenvolvimento de algoritmo paralelo, deve-se procurar escolher o estilo de paralelismo mais natural ao problema.
- Desenvolvido o algoritmo utilizando o método de programação mais natural ao estilo, pode-se conseguir resultados não satisfatórios em relação ao desempenho.
- Deve-se então transpor o algoritmo para um estilo de paralelismo mais eficiente.
- Podem-se utilizar métodos definidos na literatura para facilitar essa transição entre estilos de paralelismo.

Projeto de um Algoritmo Paralelo

- Exemplo: multiplicação de matrizes **A(*n X k*) * *B(k X m)***
  - Geométrico: pode-se determinar que cada elemento da matriz resultante seja determinado por um processador. Então, é possível a obtenção de um alto grau de paralelismo, com grande sobrecarga de comunicação em virtude da fina granulação apresentada por essa solução.

Projeto de um Algoritmo Paralelo

- *Processor Farm*: o processador mestre envia aos escravos ociosos a próxima posição da matriz produto a ser calculada. Os processadores escravo, por outro lado, enviam ao processador mestre o resultado, e tornam-se disponível para calcular um novo elemento da matriz resultante. Esta abordagem apresenta flexibilidade quanto ao número de processadores e balanceamento automático de carga. Por outro lado, a possibilidade de gargalo no processador mestre pode ser um problema.

Projeto de um Algoritmo Paralelo

- *Pipeline*: divide-se a operação de obtenção de um elemento da matriz resultado em vários estágios, cada um designado a um processador. Os estágios podem ser, por exemplo: entrada de dados, multiplicação, soma e saída do resultado. Essa organização de tarefas apresenta pouca flexibilidade e tempo de latência significativo para a obtenção de um elemento da matriz resultado.