Fontes principais

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

   ▷ Algoritmos paralelos

2. N. A. Lynch: Distributed Algorithms, Morgan Kaufmann Publishers, Inc., 96

   ▷ Algoritmos distribuídos

Coordenação distribuída

Coordenação distribuída

Em um sistema distribuído, processos podem compartilhar recursos e precisar se sincronizar.

Assim, um ambiente distribuído deve prover mecanismos para sincronização de processos, para tratamento de deadlock e para lidar com falhas.

Ordenação de Eventos

Em um sistema centralizado podemos deteminar a ordem em que 2 eventos ocorreram, pois existe um único relógio global e uma memória compartilhada.

Ordenação de Eventos

Em um sistema distribuído, algumas vezes é impossível determinar, entre 2 eventos, qual ocorreu primeiro. (Eventos são instruções executadas por uma processador instruções internas, envio e recebimento de mensagem)

Ordenação de Eventos

Processo P	Processo Q	Processo R
P_0 :	$Q_0 : envia(msg, P)$	$R_0 : envia(msg, Q)$
$P_1 : envia(msg, Q)$	$Q_1 : recebe(msg)$	$R_1 : recebe(msg)$
P_2 :	$Q_2 : envia(msg, R)$	R_2 :
P_3 :	$Q_3 : recebe(msg)$	R_3 :
$P_4 : recebe(msg)$	Q_4 :	R_4 :

Relação Aconteceu-Antes

▷ Por simplicidade, supomos que cada processo é executado em um processador distinto.

▷ Todos os eventos executados em um processador foram executados sequencialmente, logo estão totalmente ordenados.

Relação Aconteceu-Antes

▷ Pelo princípio da causalidade, um evento de recebimento de uma mensagem só é executado por um processador depois do evento de envio desta mensagem ser executado por outro processador

Relação Aconteceu-Antes

Podemos assim, definir uma relação aconteceu-antes ordenando parcialmente os eventos:

1) Se a e b são eventos executados em um mesmo processador, e a foi executado antes de b, então $a \rightarrow b$

Relação Aconteceu-Antes

2) Se a é o evento de envio de uma mensagem, executado por um processador, e b é o evento de recebimento desta mensagem, executdo por outro processador, então $a \rightarrow b$

3) Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$ (transitividade)

Relação Aconteceu-Antes

Exemplo:

1) $P_0 \rightarrow P_1$ $P_1 \rightarrow P_2$ $P_0 \rightarrow P_1$ $\cdots$
$R_0 \rightarrow R_1$ $R_0 \rightarrow R_2$ $R_1 \rightarrow R_2$ $\cdots$
$Q_0 \rightarrow Q_1$ $\cdots$

2) $P_1 \rightarrow Q_1$ $Q_0 \rightarrow P_4$ $Q_2 \rightarrow R_1$ $R_0 \rightarrow Q_3$

3) $P_0 \rightarrow Q_1$ $P_0 \rightarrow R_2$

Eventos não relacionados: P_0 e Q_0, P_0 e R_0, $R_0 e P_2$

Relação Aconteceu-Antes

A relação aconteceu-antes é uma ordenação parcial dos eventos pois nem todos os pares de eventos estão ordenados.

Existem eventos a e b que são independentes, isto é, não podemos dizer que $a \rightarrow b$ nem que $b \rightarrow a$. (Dois eventos são independentes se não existe caminho de a para b e nem de b para a)

Relação Aconteceu-Antes

Dado que 2 eventos são independentes, não interessa a ordem em que eles ocorreram. Podemos supor uma ordem qualquer. Precisamos apenas que todos os processadores suponham a mesma ordem entre estes eventos.

Podemos construir uma **ordenação total** dos eventos que respeite a relação aconteceu antes, e suponha uma ordem para os eventos independentes.

Algoritmo para construção de ordenação parcial

Cada processador terá um relógio lógico local

A cada evento executado, será atribuído um rótulo de tempo (*timestamp*)

Algoritmo para construção de ordenação parcial

Processador P

- ▷ Inicialmente $Relogio := 0$
- ▷ Ao executar um evento interno

$timestamp(evento) := Relogio$

$Relogio := Relogio + 1$

Ao executar um evento de envio de mensagem

$timestamp(evento) := Relogio$

Envia o $timestamp(evento)$ juntamente com a mensagem

$Relogio := Relogio + 1$

Ao executar um evento de recebimento de mensagem

se $Relogio \leq timestamp$(evento de envio da msg) **então**
$Relogio := timestamp$(evento de envio da msg) $+ 1$

$timestamp$(evento de recebimento) $:= Relogio$
$Relogio := Relogio + 1$

Ordenação de Eventos

Com o algoritmo visto podemos usar a ordem dos timestamps para ordenar os eventos.

Esta ordem já respeita a ordenação parcial da relação aconteceu-antes, pois se $A \rightarrow B$ então $timestamp(A) < timestamp(B)$.

Ordenação de Eventos

Precisamos ordenar alguns eventos independentes.

- ▷ Exemplo: P_3 e Q_1, onde $TS(P_3) = 3$ e $TS(Q_1) = 2$

Assim assumiremos que Q_1 aconteceu antes de P_3.

Ordenação de Eventos

Outros eventos independentes ainda não foram ordenados.

- ▷ Exemplo: P_0 e Q_0, onde $TS(P_0) = 0$ e $TS(Q_0) = 0$

Neste caso, escolher algum critério de desempate e ordená-los.

- ▷ Critério: utilizar a identificação dos processos, logo $P < Q$

Assim asumiremos que P_0 aconteceu antes de Q_0

Sejam A e B dois eventos

se $TS(A) < TS(B)$ **então**
    A aconteceu antes de B
senão se $TS(B) < TS(A)$ **então**
    B aconteceu antes de A
senão
    Seja P_i o processo onde A foi executado
    Seja P_j o processo onde B foi executado
    se $P_i < P_j$ **então**
        A aconteceu antes de B
    senão
        B aconteceu antes de A

Ordenação de Eventos

Ou seja, os eventos são ordenados pelo par

$$(TimeStamp(evento), \text{id do processo})$$

Exclusão mútua em ambiente distribuído

Exclusão mútua em ambiente distribuído

Temos n processos (identificados de 0 a $n-1$) que residem em n processadores distintos e compartilham um recurso que deve ser utilizado com exclusão mútua.

Solução Centralizada

Um processo é escolhido para ser o **coordenador**. É ele que autoriza os outros processos para usarem o recurso.

Algoritmo de um processo P

Quando P deseja usar o recurso

- P envia msg **pedido** para coordenador
- P espera receber msg **atende** do coordenador

Quando P recebe msg **atende** do coordenador

- P usa o recurso compartilhado

Quando P termina de usar o recurso

- P envia msg **libera** para coordenador

Algoritmo do Processo Coordenador

Quando coordenador recebe msg **pedido** de um processo P

se nenhum processo está usando o recurso **então**
    Coordenador envia msg **atende** para P
senão
    Coordenador coloca pedido P em uma fila

Algoritmo do Processo Coordenador

Quando coordenador recebe msg **libera** de um processo P

se fila não está vazia **então**
    Coordenador retira um pedido Q da fila
    Coordenador envia **atende** para Q
senão
    recurso fica disponível

Algoritmo do Processo Coordenador

Coordenador possui uma estrutura de dados fila onde pedidos pendente são colocados.

Coordenador pode utilizar algoritmo de escalonamento para escolher qual pedido será atendido, dentre aqueles da fila (ex. FCFS, prioridades ...)

Algoritmo do Processo Coordenador

Solução garante exclusão mútua.

Pode haver starvation dependendo do algoritmo de escalonamento escolhido.

Algoritmo do Processo Coordenador

Solução centralizada: a decisão de que o processo vai usar o recurso de cada vez é tomada de forma centralizada pelo coordenador.

Para cada vez que um processo usa o recurso, temos 3 msgs: pedido, atende, libera (custo do algoritmo)

Solução Distribuída

Utiliza mecanismo de ordenação de eventos. Cada proceso possui uma fila de pedidos pendentes.

Algoritmo de um processo P

Quando P deseja usar o recurso

- ▷ P envia msg **pedido** (junto com timestamp) para todos os demais processos
- ▷ P espera receber msg **atende** de todos os demais processos

Algoritmo de um processo P

Quando P recebe uma msg **pedido** com timestamp de um processo

se P está usando o recurso **então**
    P coloca (Pedido Q, t') na sua fila
senão se P não quer usar o recurso **então**
    P envia msg atende para Q
senão
    P compara seu pedido (Pedido P, t) com
            pedido de Q (Pedido Q, t')
    se $(t, P) < (t', Q)$ **então**
        P coloca (Pedido Q, t') em sua fila
    senão P envia msg atende para Q.

Algoritmo de um processo P

Quando P recebe msg **atende** de um dos demais processos:

se P já recebeu atende de todos os demais processos **então**
P usa o recurso

Quando P termina de usar o recurso

enquanto fila $\neq$ vazia **faça**
Retira pedido da fila $(Pedido, R, t'')$
Envia a msg atende para R.

Solução Distribuída

Observações

Algoritmo garante a exclusão mútua: Se um processo P quer usar o recurso, mas já existe um processo Q que já está utilizando o recurso, ou com maior prioridadepara usar o recurso (pedido de Q "aconteceu antes" do pedido de P) então não receberá msg atende de Q, até que Q termine de usar o recurso.

Solução Distribuída

Não há starvation: Processos usam o recurso segundo a ordem dos timestamps. Logo, se P e Q querem usar o recurso, e pedido de P tem timestamp menor, P usará o recurso primeiro. Se P deseja usar o recurso novamente, seu novo pedido terá timestamp maior, logo Q usará o recurso.

Solução Distribuída

Solução Distribuída: Todos os processos participam da decisão de qual vai usar o recurso

Custo: Para cada vez que algum processo usar o recurso temos:

- $\triangleright$ $n-1$ msgs pedido
- $\triangleright$ $n-1$ msgs atende

ou seja $O(n)$, onde n é o número de processos.

Algoritmo distribuído para determinar componentes conexos de um Grafo

Algoritmo sequencial

Entrada:

- ▷ $nVertices$: número de vértices do grafo
- ▷ mat_adj: matriz de adjacência

Saída:

- ▷ $componente$: vetor com uma posição para cada vértice.

Se vértices i e j estão com um mesmo componente, então $componente[i] = componente[j]$

Algoritmo sequencial

▷ Inicia cada vértice como um componente

para $i := 0$ **até** $nVertices - 1$ **faça**

$componente[i] := i$

▷ Investiga vértices percorrendo parte triangular da matriz

para $i := 0$ **até** $nVertices - 1$ **faça**

para $j := 0$ **até** $i - 1$ **faça**

▷ Se existe aresta (i, j) e i e j ainda não estão

▷ no mesmo componente

se $mat\_adj[i, j] = 1$ e $componente[i] \neq componente[j]$ **então**

se $componente[i] < componente[j]$ **então**

$comp\_menor = componente[i]$

$comp\_maior = componente[j]$

senão

$comp\_menor = componente[j]$

$comp\_maior = componente[i]$

para $k := 0$ **até** i **faça**

se $componente[k] = comp\_maior$ **então**

$componente[k] = comp\_menor$

Algoritmo Distribuído

Idéia

Cada processo é responsável por uma parte dos vértices e investiga parte da matriz de adjacência correspondente a estes vértices

Quando um processo muda um componente de um vértice, ele informa alguns dos demais processos (apenas aqueles que poderiam alterar seus vértices com esta informação)

Algoritmo Distribuído

Ao receber esta informação, ele atualiza suas estruturas e investiga os vértices alocados a ele, em relação ao vértice alterado.

Caso este processador muda o componente ...

O algoritmo prossegue assim, até chegar ao fim, quando não existem mais informação a serem enviadas ou recebidas.

Estrutura de dados de cada processo

Entrada:

- ▷ $nVertices$: número de vértices do grafo
- ▷ mat_adj: matriz de adjacência
- ▷ $nprocessadores$: número de processadores

Saída:

- ▷ $componente[nVertices]$

Estrutura de dados de cada processo

Estruturas auxiliares:

- $vini$: vértice inicial
- $vfim$: vértice final
- $nvertices\_proc$: número de vértices por processadores
- id_proc: $0, 1, 2, \cdots, nprocessadores - 1$

Algoritmo de cada processo

▷ Inicia cada vértice como um componente

para $i := 0$ **até** $vfim$ **faça**

$componente[i] := i$

▷ Investiga vértices alocadados a este processador

para $i := vini$ **até** $vfim$ **faça**

para $j := 0$ **até** $i - 1$ **faça**

▷ Investiga vértice i em relação a vértice j

$investigaVertice(i, j, i)$

$terminou := false$
enquanto não $terminou$ **faça**
▷ Espera receber msg de algum dos demais processadores
$recebe(msg)$
se $msg = (NOVO\_COMP, v, comp\_v)$ e
$comp\_v < componente[v]$ **então**
$componente[v] := comp\_v$
▷ investiga vértices alocados a
▷ este processador em relação a v
para $i := vini$ **até** $vfim$ **faça**
$investigaVertice(i, v, vfim)$

função $Investiga\_Vertice$(vertice i, vertice j, vertice $final$)

▷ Se existir aresta (i, j) e vértices i e j ainda não

▷ estão no mesmo componente

se $mat\_adj[i - vini, j] = 1$ e $componente[i] \neq componente[j]$ **então**

▷ junta componentes

se $componente[i] < componente[j]$ **então**

$comp\_menor := componente[i]$

$comp\_maior := componente[j]$

senão

$comp\_menor := componente[j]$

$comp\_maior := componente[1]$

▷ continua no proximo slide ...

▷ Se existir aresta (i, j) e vértices i e j ainda não
▷ estão no mesmo componente
se $mat\_adj[i - vini, j] = 1$ e $componente[i] \neq componente[j]$ **então**
...
para $k := 0$ **até** $final$ **faça**
se $componente[k] = comp\_maior$ **então**
$componente[k] := comp\_menor$
▷ Determine processador responsável por vértices k
$proc\_alocado := (k\ div\ nverticesProc) + 1$
▷ Envia msg com informação de novo
▷ componente de k, para todos os
▷ processadores a partir de
proc_alocado, exceto para este processador
para $t := prox\_alocado$ **até** $nprocessadores - 1$ **faça**
se $t \neq id\_proc$ **então**
Envia para processador t
$msg(NOVO\_COMP, k, comp\_menor)$

Algoritmo distribuído para determinar componentes conexos de um Grafo

Observações:

- ▷ Topologia completamente conexa.
- ▷ Algoritmo deve ter distribuição inicial dos dados ($nvertices$ e mat_adj) e recolhimento final dos dados (componente).
- ▷ Carga não é balanceada: É possível melhorar.

Algoritmo distribuído para determinar componentes conexos de um Grafo

▷ Não determinismo: durante o algoritmo, um processador pode receber msgs dos demais processadores em uma ordem independente.

▷ Precisa de bufferização (capacidade de comunicação limitada, maior que 0): criação de um buffer recebimento em cada processador,que é concorrente com o processo que determina os componentes conexos

▷ Precisa de detecção de terminação.

Detecção de terminação

▷ A detecção de terminação poder ser feita de forma centralizada: apenas o processo de $id = 0$ detecta a terminação e avisa aos demais processos.

Detecção de terminação

▷ Para determinar se não há mensagens em trânsito, cada processo mantém informações sobre o número de mensagens que ele enviou para cada um dos demais processos, e o número de mensagens que ele recebeu de cada um dos demais processos.

Detecção de terminação

▷ Quando um processo (exceto o processo 0) está ocioso (isto é, não possui trabalho para realizar no algoritmo), ele envia estas informações para o processo 0.

▷ O processo 0 possui então as informações de cada processo, inclusive dele mesmo.

Detecção de terminação

▷ Quando o processo 0 está ocioso, ele compara estas informações para determinar se a terminação já foi atingida.

Se o número de mensagens enviadas por P_i para P_j é igual ao número de mensagens que P_j recebeu de P_i, para todo par de processos P_i e P_j, $P_i \neq P_j$, então a terminação foi atingida.

Fim