Programação Concorrente e/ou Paralela

Profa. Inês de Castro Dutra

COPPE Sistemas/UFRJ

Objetivos

- Cobrir os conceitos principais em programação concorrente
- Habilitação para desenvolvimento e pesquisa nesta área

Pré-Requisitos

- Arquitetura I
- Sistemas operacionais I
- Disposição para estudar e programar

Avaliação

- Projetos práticos (sem adiamentos)
- Apresentação de artigos e capítulos selecionados
- Prova(s) (depende da turma)
- Testes surpresa, se necessário

Material Didático

- Andrews, "Concurrent Programming: Principles and Practice"
- Foster, "Designing and Building Parallel Programs"
- Wolfe, "High Performance Compilers for Parallel Computing"
- Hwang, "Advanced Computer Architecture"
- Culler, "Computer Architecture"
- Alguns artigos científicos
- `http://www.cos.ufrj.br/~ines/courses/progpar.html`

Ementa

1. Introdução
2. Modelos de programação
3. Arquiteturas/máquinas //s
4. Sincronização
5. Passagem de mensagens
6. Construções e técnicas em programação //
7. Linguagens e sistemas runtime para prog //
8. Considerações de desempenho

Introdução

- Por que paralelismo?
- Dimensoes da programação //
- Um pouco de história
- Projeto e verificação de programas //s

Por que Paralelismo?

- Limites físicos
- Paralelismo natural
- Software de sistema
- Interesse intrínseco

Aplicações: científicas, sistemas de transação, sistemas de tempo real, tratamento de interrupções

Dimensões do Paralelismo

- Processos e threads
- Modelos de programação
- Concorrente x // x distribuído
- Sistemas //s e distribuídos
- Arquiteturas //s
- Linguagens e runtime
- Métricas de desempenho

Processos e Threads

Processo $\rightarrow$ espaço de endereçamento

Thread $\rightarrow$ mesmo espaço que pai

Processo (lógico) != processador (físico)

Processo abstração de processador

Modelo Von Neumann $\rightarrow$ 1 fluxo de controle

Progs concorrentes $\rightarrow$ +1 fluxo

Modelos de Programação

Define interface usada pelo programador

Paralelismo, comunicação, sincronização, etc

Exemplos: sequencial, memória compartilhada, passagem de mensagens

Concorrente x Paralelo x Distribuído

Concorrente: +1 fluxo de controle

Paralelo: concorrente onde comunicação via memória compartilhada

Distribuído: concorrente onde comunicação via passagem de mensagens

Sistemas Paralelos e Distribuídos

Paralelo: espaço de endereçamento único em hardware

Distribuído: múltiplos espaços de endereçamento

Possível rodar progs distribuídos em sistemas //s e vice-versa

Para simplificar, diremos que todos os progs e sistemas são "paralelos" nesse curso

Arquiteturas Paralelas

Arquitetura determina conj de construções e técnicas de programação que podem ser implementadas eficiente/

Mais comuns: SIMD e **MIMD**

Exs: Power Challenge, KSR-1, Sequent, Origin, T3E, Paragon, CM-5. Dash, Flash, Alewife, Typhoon, Enterprise, Multiplus, NCP I, NCP2.

Linguagens, Compiladores e Bibliotecas

Vants lings: sintaxe especial, efeitos colaterais e contexto implícito, integração (verificação de tipos, threads, tratamento de exceções, etc)

Vant compiladores: simplicidade do modelo

Vants biblio: facilidade de modificação, uso com lings existentes, uso com múltiplas lings (principalmente passagem de msgs)

Métricas de Desempenho

Amdahl's Law:

Speedup s = T(1)/T(p)

Trabalho total c = T_s + T_p = T(1)

T(p) = T_s + T_p/p

s = (T_s + T_p) / (T_s + T_p/p) =
= c / (T_s + T_p/p) $\rightarrow$ c/T_s qdo p $\rightarrow$ inf

Um Pouco de História

Computadores inicial/ eram single user – anos 50

1a motivação: devices de E/S; espera ocupada desperdício

Timeslicing (batch) → multiprogramação (concorrência)

E/S programável → condições de corrida (//ismo restrito)

Interrupções de E/S assíncronas – meados dos anos 60

Comunicação entre processos de usuário – início dos 70

Redes (processa/ distribuído) – início dos 70

Multiprocs baseados em mem multi-ported – fim dos 70

Multicomps e multiprocs baseados em barramento – início dos 80

Multiprocs escaláveis – meados dos anos 80

Multiprocs com coerência em HW – início dos 90

Projeto e Verificação de Progs //s

Projeto de algoritmos //s fora do escopo

Esse tópico e vários relacionados vistos em outros cursos/livros

Importante: garantir liveness e safety

Liveness: coisas boas eventualmente acontecem

Safety: coisas más nunca acontecem

Exs liveness: nenhum processo espera para sempre, o prog termina

Exs safety: exclusão mútua, não acontece overflow de buffers

Modelos de Programação + Comuns

- Sequencial
- Memória compartilhada
- Passagem de mensagens
- SPMD vs. MPMD ou //ismo de dados vs. tarefas

Outros Modelos de Programação

- Linda
- Actors
- Dataflow
- Logic
- Functional
- Constraints

Modelo de Programação Sequencial

Modelo mais simples

//ismo implementado pelo compilador ou software + básico

```
for i = 1 to N
    a[i] = 1
```

Exs: HPF (quase) e outros Fortrans (compilador); Andorra-I e outras linguagens declarativas (runtime)

Modelo de Memória Compartilhada

Modelo mais complexo, mas ainda próximo do sequencial

//ismo implementado pelo programador através da linguagem ou chamadas ao software + básico

Sincronização necessária

Comunicação transparente (implementada por sw ou hw)

Exs: SR (linguagem), TreadMarks (runtime), SoftFLASH (SO)

Modelo de Memória Compartilhada

```
doall i = 1 to N
    a[i] = 1

for j = 1 to NPROCS-1
    fork(compute,j)
compute(0)

lock(mutex)
    x = x + 1
unlock(mutex)
```


Modelo de Passagem de Mensagens

Modelo extremamente complexo

//ismo implementado pelo programador através da linguagem ou chamadas ao software + básico

Comunicação explícita através de passagem de msgs

Sincronização geralmente associada às msgs

Exs: SR e Occam (linguagem), PVM e MPI (runtime)

Modelo de Passagem de Mensagens

```
Proc pid:

chunk = N/NPROCS
for j = pid*chunk to (pid+1)*chunk-1
    a[i] = 1
send(dest,&a[pid*chunk],chunk*sizeof(int))
```


Outros Modelos

Linda

- + complexo que compart, mas + simples que pass de msgs
- Híbrido: espaço de tuplas + comunic explícita + sincr implícita
- Comandos de acesso a tuplas: in (receive), out (send), rd (consulta), eval (criação de processos), inp e rdp (acessos não bloqueantes)
- Primitivas de Linda podem ser adicionadas a lings tradicionais

Exemplo: SOR

- Computação sobre uma matriz
- Grupo de linhas consecutivas para cada proc
- Cada elemento calculado usando vizinhos
- Comunicação nas bordas

Exemplo: SOR

Modelo sequencial

```
for num_iters
    for num_linhas
        compute
```

Modelo de memória compartilhada

```
for num_iters
    for num_linhas in //
        compute
```

ou

```
for num_iters
    for num_minhas_linhas
        compute
    barreira
```

Exemplo: SOR

Modelo de passagem de msgs (send nao bloqueante)

```
define submatriz local
for num_iters
    if pid != 0
        envia primeira linha a pid-1
        recebe limite superior de pid-1
    if pid != P-1
        envia ultima linha a pid+1
        recebe limite inferior de pid+1
    for num_linhas
        compute
```


Comparação de Modelos

- Sequencial ideal, mas depende de software sofisticado
- Mem compart leva a programas mais simples, mas sincr explícita
- Pass de msgs leva a comunicação eficiente e sincr implícita, mas dificulta programação

SPMD vs. MPMD

- Classificação a nível de programas
- SPMD = //ismo de dados = prog p/ SIMD rodando em MIMD
- MPMD = //ismo de tarefas; Mestre/escravo exemplo

Tópicos Clássicos em MC

Condições de corrida

- Corrida ocorre qdo ações não são sincronizadas e comporta/ depende da ordem das ações
- Corrida às vezes não causa problema, exs: mestre/escravo ou fila de tarefas
- Em geral, devemos evitar corridas

Exemplo de corrida a evitar

```
  Proc 1          Proc 2
load X,reg      load X,reg
inc reg         inc reg
store reg,X     store reg,X
```

Tópicos Clássicos em MC

Sincronização

- Usada para evitar corridas
- Dois tipos: exclusão mútua e sincr por condição
- Precisa de instruções atômicas
- Cuidado para nao sincronizar demais

Exemplo de sincronização

```
  Proc 1          Proc 2
mutex L         mutex L
load X,reg      load X,reg
inc reg         inc reg
store reg,X     store reg,X
demutex L       demutex L
```


Sincronização

- Escritas e leituras
- Read-modify-write ou fetch-and-phi
- Sincr por condição fácil com leituras e escritas
- Exclusão mútua difícil com leituras e escritas apenas

Sincr por condição: "var X contem valor Y"
$\rightarrow$ ler X até valor desejado

Exclusão: Dekker início dos anos 60 (solução para 2 procs); Dijkstra 1965 (solução para N procs); Peterson 1981 (solução + simples para 2 procs)

Sincronização

Ler repetidamente uma var até certo valor = espera ocupada

Espera ocupada gasta ciclos preciosos

Necessário que sincr interaja com escalonador para bloquear: semáforos e monitores

Tradeoff: spin qdo tempo esperado menor que overhead de re-escalonamento

Tópicos Clássicos em PM

- Comunicação bloqueante vs. nao-bloqueante
- Naming e comunicação coletiva
- Overhead de messaging

Com. Bloqueante e Nao-Bloqueante

- Com. bloqueante nao requer buffers
- Com. nao-bloqueante $\rightarrow$ max concorrência; fluxo e erros problemas
- Send bloqueante espera que receptor esteja pronto
- Receive bloqueante espera até que exista msg
- Send nao-bloqueante completa imediata/, exceto qdo nao há buffer
- Receive nao-bloqueante completa imediata/

Naming e Comunicação Coletiva

- Canal, porta, ou processo usados para especificar receptor em comunicação 1-pra-1
- Outras formas para comunicação coletiva 1-pra-vários, vários-pra-1 e vários-pra-vários
- Exs: links em Demos, Demos-MP, and Arachne; espaço de tuplas em Linda

Overhead de Messaging

- Passagem de msgs geralmente custosa (feita em sw e intervenção do SO)
- Sistemas modernos evitam o SO (apenas mapeamento e verificação de proteção)
- Exs: Active msgs, Fast msgs, API da Myricom

Próxima Aula

Capítulo 2 do livro do Andrews

Discussão em aula

Aluno responsável por esclarecer dúvidas

Revisão

- Por que //ismo?
- Dimensões da programação //
- Um pouco de história
- Projeto e verificação de programas //s
- Modelos + comuns
- Tópicos clássicos em memória compartilhada
- Tópicos clássicos em passagem de mensagens

Hoje

Tópicos em Compiladores //izadores

- Tópicos em linguagens
- Relações de dependência de dados
- Re-estruturação de laços
- Análise de concorrência

Decomposição e distribuição de dados (HPF)

	P0	P1	P2	P3
	x(1)	x(4)	....	....
block	x(2)	x(5)		
	x(3)	....		

	P0	P1	P2	P3
	y(1)	y(2)	y(3)	y(4)
cyclic	y(5)	y(6)	....	....
	y(9)	....		

	P0	P1	P2	P3
	z(1)	z(3)	z(5)	....
cyclic	z(2)	z(4)	z(6)	
(2)	....	....	....	

Diferentes tipos de laços:

Atribuição de arrays – Ex: a(1:n) = b(0:n-1)*2 + c(2:n+1)

do (seq) – uma iter só comeca depois que a precedente termina

dopar (par) – iters executadas por procs diferentes e dados são mesmos que qdo laço iniciado

doall (dopar especial) – nao há dependências entre iters

doacross (par) – existem dependências e atribuições de uma iter valem para outras

Relações de Dependência

Relações são usadas para representar as restrições de ordenação entre comandos em um prog

Ex: Mover (2) acima de (1) ou mudar a ordem de (3) e (4) muda a semântica. Mudar a ordem de (2) e (3) não causa problema.

```
(1) A = 0
(2) B = A
(3) C = A + D
(4) D = 2
```


Relações de Dependência

Grafo de dependência de dados: vértices = comandos ou blocos, arestas = restrições

Restrições:

- Dep de fluxo: var atribuída em um comando e usada num seguinte
- Anti-dep: var usada num comando e atribuída num seguinte
- Dep de saída: var atribuída num comando e re-atribuída num seguinte

Exemplo

```
(1) A = 0        S1 --+
(2) B = A        |    |
(3) C = A+D      V    | fluxo
(4) D = 2        S2   |
                      |
                 S3 <-+
                 |
                 - anti
                 |
                 V
                 S4
```

Grafo de precedência: grafo de dep sem ciclos

Dependência em Laços Sequenciais

Dependências "carregadas" por laços: dep entre instâncias de comandos em iters diferentes

Dependências "independentes" de laços: dep entre instâncias na mesma iter

Forward (backward) dep: fonte precede o destino (destino precede fonte)

Exemplo

```
    (1) do I=2,9
    (2)    X[I] = Y[I] + Z[I]
    (3)    A[I] = X[I-1] + 1
    (4) enddo
```

```
Rela\c c\~oes de dep causadas por X:

        I=2             I=3
(2) X[2]=Y[2]+Z[2]  X[3]=Y[3]+Z[3]
(3) A[2]=X[1]+1     A[3]=X[2]+1

Forward dep carregada de (2) para (3):

    S2
    |
    | (1)
    V
    S3
```


Espaco de Iterações

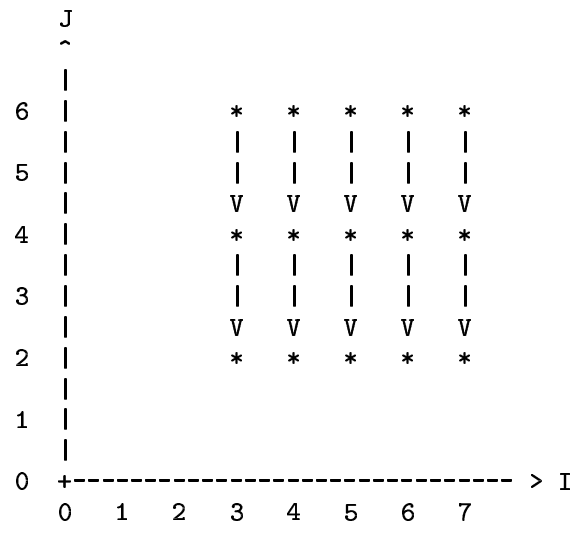
1 ponto para cada iteração

Aresta de fonte para destino se comando em uma iter depende de comando em outra iter (grafo de dependência entre iters)

Grafo nao pode ser construído, portanto grafo de dep anotado (como a dep cruza as iterações no espaco). Ex: (1) na aresta S2 $\rightarrow$ S3 do exemplo anterior

Exemplo

```
(1) do I=3,7
(2)    do J=6,2 by -2
(3)       A[I,J] = A[I,J+2] + 1
(4)    enddo
(5) enddo
```



Dependência em Laços Paralelos

Dois comandos ou duas iters têm um "conflito" qdo podem se referir a mesma posição de memória

Conflitos têm que ser resolvidos para relações de dep para que compilador garanta semântica correta

Lista de comandos: conflitos resolvidos completando primeiro acesso antes de iniciar segundo, i.e. conflitos resolvidos como anti-deps

Laço: conflito resolvido de acordo com regras do laço

Dependência em Laços Paralelos

Num do conflitos entre iterações são resolvidos completando o acesso na iteração anterior primeiro

Num dopar valores computados numa iter nao podem ser usados por outra $\rightarrow$ conflitos resolvidos como anti-deps ou deps de saída

Num doall nao ha dependências entre iters $\rightarrow$ conflitos resolvidos como relações independentes de laço

Exemplo

```
(1) dopar I=2,20
(2)    X[I] = Y[I] + 1
(3)    Z[I] = X[I-1] + X[I] + X[I+1]
(4) enddopar
```

```
      I=2                  I=3
(2) X[2]=Y[2]+1          X[3]=Y[3]+1
(3) Z[2]=X[1]+X[2]+X[3]  Z[3]=X[2]+X[3]+X[4]
```

Dep de fluxo entre (2) e (3), distância (0)

Anti-dep entre iterações com distância (-1) e (1)

Re-Estruturação de Laços

Usada para re-ordenar a execução de comandos e iterações. Legal quando preserva a semântica

Várias técnicas:

- Peeling
- Splitting
- Scalar expansion
- Fusion
- Fission
- Interchanging
- Strip mining
- Tiling
- Unrolling

Peeling

Coloca 1a(s) ou última(s) iter(s) do laço em código separado

Usado p/ ajustar num iters (para permitir fusion) ou p/ remover um comando condicional que testa a var de índice

```
Antes:  do I=1,N
            A[I] = (X + Y) * B[I]
        enddo

Depois: if N >= 1 then
            A[1] = X + Y * B[1]
            do I=2,N
                A[I] = X + Y * B[I]
            enddo
        endif
```

Splitting

Divide o conj de índices em duas partes

Usado pelos mesmos motivos que peeling

```
Antes:  do I=1,100
            A[I] = B[I] + C[I]
            if I > 10 then
                D[I] = A[I] + A[I-10]
            endif
        enddo

Depois: do I=1,10
            A[I] = B[I] + C[I]
        enddo
        do I=11,100
            A[I] = B[I] + C[I]
            D[I] = A[I] + A[I-10]
        enddo
```


Scalar Exp, Fusion, Fission, Interchanging

SE: escalares são promovidos para arrays para quebrar anti deps e deps de saída

Fusion: funde 2 laços adjacentes com mesmos limites num único laço. Usado p/ reduzir custo de teste e desvio, melhorar localidade temporal e permitir otimizações escalares como eliminação de subexpressões

Fission: inverso de fusion. Usado p/ melhorar taxas de acerto em caches

Interchanging: troca laços aninhados. Usado p/ reduzir custo de início de laços, ajudar na descoberta automática de //ismo (mover laços sem dependências p/ dentro) e melhorar localidade temporal e espacial

Strip Mining

Decompõe um laço em dois laços aninhados

Usado para melhorar a localidade de dados

```
Antes:  do I=1,N
            A[I] = B[I] + C[I]
        enddo

Depois (strip de tam s):
        do Is=1,N by s
             do I=Is,min(N,Is+s-1)
                 A[I] = B[I] + C[I]
             enddo
        enddo
```


Tiling e Unrolling

Tiling: similar a strip mining, mas p/ laços aninhados. Usado p/ criar versões "blocadas" de programas que têm melhor localidade de dados

Unrolling: desenrola laços. Usado p/ aumentar a quantidade de paralelismo de instruções

Análise de Concorrência

Paradigma mestre/escravo: parte principal do prog executada por apenas 1 proc (mestre). Qdo região paralela alcançada, mestre inicia múltiplos escravos paralelos. Qdo escravos completam execução, sincronizam através de barreira.

Implementação:

- Escravos: processos ou threads?
- Mestre participa da execução paralela?
- Mestre diferenciado? Ou último escravo na barreira vira mestre?

Escalonamento

Estático: N/P iters consecutivas por escravo (blocos) ou uma iter por escravo em round-robin (cíclico). Estático não funciona bem qdo existe desbalanceamento de carga

Dinâmico:

- Self-scheduling: cada escravo pega uma iter numa fila
- Guided self-scheduling: 1/P tarefas de cada vez
- Affinity scheduling: cada escravo pega as iters que já executou (laço paralelo dentro de sequencial)

Sincronização

Quando iters não são independentes de laco, sincronização pode ser necessária

Sincronização implementada através de seções críticas ordenadas (SCOs)

SCOs = seções críticas nas quais procs entram em ordem pré-determinada

SCOs usadas para satisfazer deps de dados carregadas por laços

SCOs são implementadas através de primitivas await e advance

Exemplo

```
do I=2,n
    D[I] = D[I-1] + A[I]
enddo
```

Cada escravo:

```
await(I-1)          await e advance podem ser
fetch D[I-1]->r1    implementados dif formas.
fetch A[I]->r2      Ex vetor de bits (1/iter);
add r1, r2->r3      advance seta i-esimo bit,
store r3->D[I]      await espera ate que
advance             (i-1)-esimo bit setado.
```

Note que escalonamento de blocos não deve ser usado para laços paralelos com sincronização!

Código Paralelo para Laços Sequenciais

Qdo iters independentes $\rightarrow$ sem problema

Qdo existem deps carregadas $\rightarrow$ 3 opções:

- Remover deps (re-estruturação dos laços)
- Sincronização através de SCOs
- Código sequencial

Deps carregadas podem ser eliminadas por fission, strip-mining ou striping.

Exemplo

```
do I=1,N
    A[I+8] = A[I] + B[I]
    B[I+5] = C[I] + 1
enddo

Depois de strip-mining,
    + interno pode rodar //

    do IS=1,N step 5
        do I=IS, min(N,IS+4)
            A[I+8] = A[I] + B[I]
            B[I+5] = C[I] + 1
        enddo
    enddo
```

Código Paralelo para Laços Paralelos

Código p/ doall fácil de gerar; p/ doacross precisa sincronização

Código p/ dopar com deps. Ex:

```
dopar I=2:N-1
    A[I] = (A[I-1] + A[I+1])/2
enddopar
```

Como dopar resolve conflitos como anti-deps:

```
doall I=2:N-1
    T1[I] = A[I-1]
    T2[I] = A[I+1]
enddoall
doall I=2:N-1
    A[I] = (T1[I] + T2[I])/2
enddoall
```


Código Paralelo para Laços Paralelos

Tiling pode ser usado para reduzir overheads de sincronização e escalonamento em laços aninhados

Re-estruturação pode ser usada para melhorar padrão de acesso à memória, eliminar sincronização e aumentar a granularidade das tarefas //s

Próxima Aula

Capítulos 13 e 14 do livro do Wolfe

Discussão em aula

Alunos responsáveis por esclarecer dúvidas