MAC0438 - Programação Concorrente

Daniel Macêdo Batista

IME - USP, 5 de Março de 2013

Roteiro

Lembretes

Continuação do exemplo do grep

Classes de aplicações

Paradigmas de concorrência

Processos e sincronização

Paralelizando outro algoritmo

Ações atômicas

Sintaxe

Lembretes

O que eu estava devendo e o que ainda estou devendo

- Enunciados dos EPs
  - EP1 OK
  - EP2 e EP3 Pendente – Vou divulgar mais para frente
- Apresentar o monitor Pendente – Não depende de mim :(

O que vocês estavam devendo

- Datas confirmadas:
  - Provas normais: 26/04 e 28/06
  - Sub e Rec: 05/07 e 12/07
  - EPs: 01/04, 06/05 e 17/06

Continuação do exemplo do grep

Buscando padrões em um arquivo em // (Tentativa 4)

- Utilizando `while` dentro de `co` ao invés de `co` dentro de `while`

```
string buffer;
bool done = false;

co # Primeiro processo. Busca padroes
   string linha1;
   while (true) {
      espere ate buffer estar cheio ou done ser verdade;
      if (done) break;
      linha1=buffer;
      sinalize que buffer esta vazio;
      busque por padrao em linha1;
      if (padrao em linha1)
            write linha1;
   }
```

Buscando padrões em um arquivo em // (Tentativa 4)


```
// # Segundo processo. Leitura de linhas novas
   string linha2;
   while (true) {
      leia a proxima linha de stdin em linha2;
      if (EOF) {
         done=TRUE;
         break;
      }
      espere ate buffer estar vazio;
      buffer=linha2;
      sinalize que buffer esta cheio;
   }
oc;
```

Classes de aplicações

Multithreaded

- Geralmente mais processos do que processadores
- Gerenciamento de janelas
- Sistemas operacionais para múltiplos processadores e de tempo compartilhado
- Sistemas de tempo real

Computação distribuída

- Troca de mensagens
- Servidores de arquivos
- Servidores web

Computação paralela

- Acelerar a execução de um processo pesado
- Computações científicas
- Processamento de imagens

Paradigmas de concorrência

Paralelismo iterativo


```
double a[n,n], b[n,n], c[n,n];

for [i = 0 to n-1] {
  for [j = 0 to n-1] {
    # compute inner product of a[i,*] and b[*,j]
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Sequential Matrix Multiplication

Paralelismo iterativo


```
co [i = 0 to n-1] {  # compute rows in parallel
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication by Rows

Paralelismo iterativo


```
co [j = 0 to n-1] {  # compute columns in parallel
  for [i = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication by Columns

Paralelismo iterativo


```
co [i = 0 to n-1, j = 0 to n-1] { # all rows and
  c[i,j] = 0.0;                   # all columns
  for [k = 0 to n-1]
    c[i,j] = c[i,j] + a[i,k]*b[k,j];
}
```

Parallel Matrix Multiplication by Rows and Columns

Paralelismo iterativo


```
co [i = 0 to n-1] {     # rows in parallel then
  co [j = 0 to n-1] {   # columns in parallel
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication Using Nested co Statements

Paralelismo iterativo


```
process row[i = 0 to n-1] { # rows in parallel
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[ij] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication Using a Process Declaration

Paralelismo iterativo


```
process worker[w = 1 to P] {    # strips in parallel
  int first = (w-1) * n/P;      # first row of strip
  int last = first + n/P - 1;   # last row of strip
  for [i = first to last] {
    for [j = 0 to n-1] {
      c[i,j] = 0.0;
      for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
  }
}
```

Parallel Matrix Multiplication by Strips (Blocks)

Paralelismo recursivo


```
double fleft = f(a), fright, area = 0.0;
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width] {
  fright = f(x);
  area = area + (fleft + fright) * width / 2;
  fleft = fright;
}
```

Iterative Quadrature Program

Paralelismo recursivo


```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # recurse to integrate both halves
    larea = quad(left, mid, fleft, fmid, larea);
    rarea = quad(mid, right, fmid, fright, rarea);
  }
  return (larea + rarea);
}
```

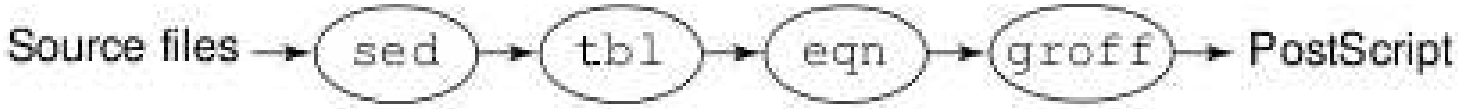
Recursive Procedure for Quadrature Problem

Paralelismo recursivo

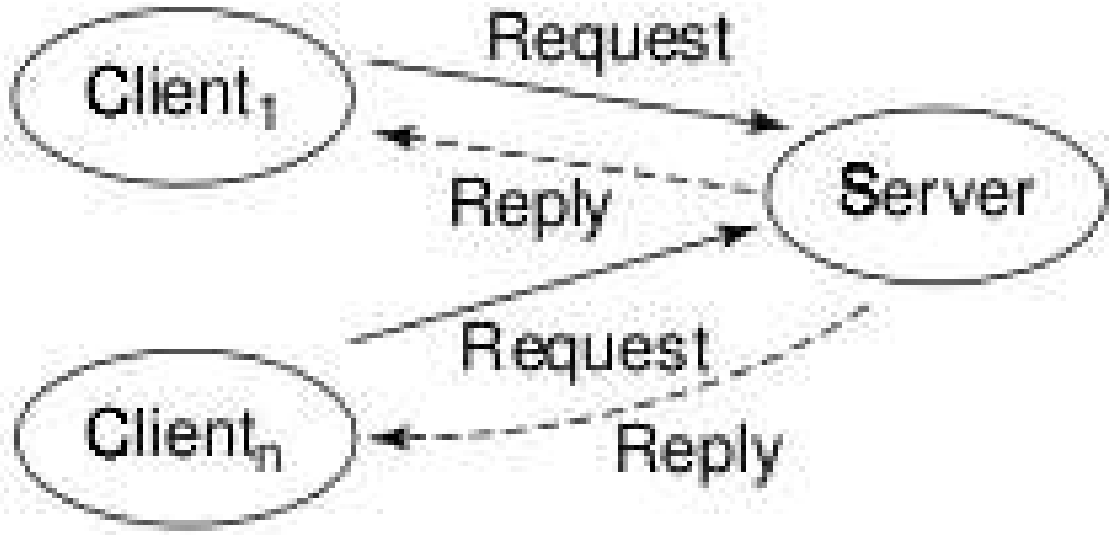

```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # recurse to integrate both halves in parallel
    co larea = quad(left, mid, fleft, fmid, larea);
    // rarea = quad(mid, right, fmid, fright, rarea);
    oc
  }
  return (larea + rarea);
}
```

Recursive Parallel Adaptive Quadrature

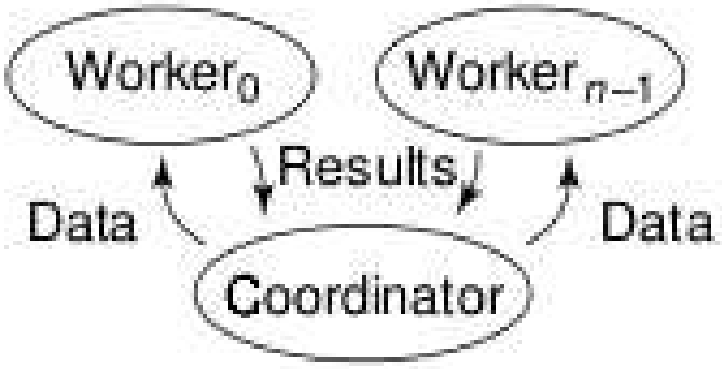
Produtores e consumidores (pipeline)



Clientes e servidores



Interação entre peers



(a) Coordinator/worker interaction

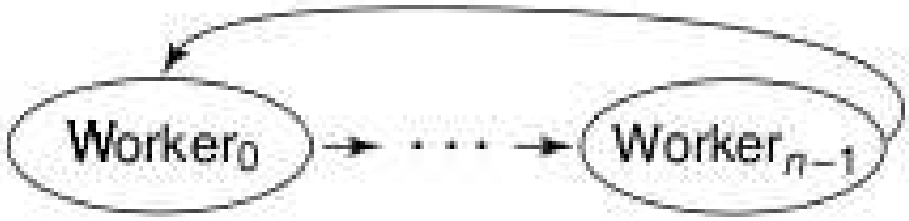
Interação entre peers


```
process worker[i = 0 to n-1] {
  double a[n];     # row i of matrix a
  double b[n,n];   # all of matrix b
  double c[n];     # row i of matrix c
  receive initial values for vector a and matrix b;
  for [j = 0 to n-1] {
    c[j] = 0.0;
    for [k = 0 to n-1]
      c[j] = c[j] + a[k] * b[k,j];
  }
  send result vector c to the coordinator process;
}

process coordinator {
  double a[n,n];   # source matrix a
  double b[n,n];   # source matrix b
  double c[n,n];   # result matrix c
  initialize a and b;
  for [i = 0 to n-1] {
    send row i of a to worker[i];
    send all of b to worker[i];
  }
  for [i = 0 to n-1]
    receive row i of c from worker[i];
  print the results, which are now in matrix c;
}
```

Matrix Multiplication Using Coordinator/Worker Interaction

Interação entre peers



(b) A circular pipeline

Processos e sincronização

Como coordenar o acesso às variáveis?

- Sincronização
  - Exclusão mútua
  - Sincronização por condição

Estado de um programa concorrente

- Um conjunto com os valores de todas as variáveis (explícitas e implícitas)
- O programa sempre começa em um estado inicial
- Cada processo executa de forma independente, variando o estado
- No exemplo abaixo, quais os estados (variáveis explícitas)?

```
int x=4;

co x++;
// x=1;
oc
```

Ações atômicas

- Ações indivisíveis que verificam ou mudam o estado do programa
- Cada comando do processo é um conjunto de ações atômicas
- No exemplo abaixo, quantas ações atômicas cada processo possui?

```
int x=4;

co x++;
// x=1;
oc
```

Histórias

- A execução de um programa concorrente corresponde à intercalação de ações atômicas
- Uma execução particular é uma história
  - $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots s_n$
- Uma história pode ser simulada por um programa sequencial

Histórias

- A cada estado, qualquer ação atômica de qualquer processo pode executar
- Quantidade imensa de histórias (sem esquecer de quando os processos tem condicionais)

Para que sincronização?

- Restringir as possíveis histórias pois nem todas são desejáveis
- Exclusão mútua usa ações atômicas de nível mais baixo para transformar o acesso à seção crítica em uma ação atômica
- Sincronização por condição atrasa a execução de um processo até o momento que ele possa acessar uma variável compartilhada

Paralelizando outro algoritmo

Buscando o máximo de um vetor

- Um vetor a[n] de inteiros positivos

```
int m=0;
```

Buscando o máximo de um vetor

- Um vetor a[n] de inteiros positivos

```
int m=0;
for [i=0 to n-1] {
```

Buscando o máximo de um vetor

- Um vetor a[n] de inteiros positivos

```
int m=0;
for [i=0 to n-1] {
  if (a[i] > m)
     m=a[i];
}
write (m);
```

- Ideias para paralelizar?

Buscando o máximo de um vetor em // (Tentativa 1)

- Examinar cada posição do vetor em paralelo

```
int m=0;
co [i=0 to n-1] {
  if (a[i] > m)
     m=a[i];
}
write (m);
```

- Está certo? (Quantas possíveis saídas tem o programa?)

Buscando o máximo de um vetor em // (Tentativa 1)

- Examinar cada posição do vetor em paralelo

```
int m=0;
co [i=0 to n-1] {
  if (a[i] > m)
     m=a[i];
}
write (m);
```

- O problema é a leitura e a escrita de m de forma separada. Como unir?

Buscando o máximo de um vetor em // (Tentativa 2)

- Combinar as ações de leitura e escrita de m

```
int m=0;
co [i=0 to n-1] {
  <if (a[i] > m)
     m=a[i];>
}
write (m);
```

- O `if` e a atribuição agora são indivisíveis
- Está certo? (Quantas possíveis saídas tem o programa?)
- É eficiente?
- Em tempo de execução, qual a diferença para o problema original sequencial?

Buscando o máximo de um vetor em // (Tentativa 3)

- Temos que proteger as atualizações dos valores de m
- Ideias?

Buscando o máximo de um vetor em // (Tentativa 3)

- Temos que proteger as atualizações dos valores de m

```
int m=0;
co [i=0 to n-1] {
  if (a[i] > m)
     <m=a[i];>
}
write (m);
```

- Está certo? (Quantas possíveis saídas tem o programa?)

Buscando o máximo de um vetor em // (Tentativa 4)

- Temos que realizar as comparações em paralelo (somente leitura)
- Temos que garantir que quando m é atualizado, de fato está sendo visto o maior valor
- Ideias?

Buscando o máximo de um vetor em // (Tentativa 4)

- Temos que realizar as comparações em paralelo (somente leitura)
- Temos que garantir que quando m é atualizado, de fato está sendo visto o maior valor

```
int m=0;
co [i=0 to n-1] {
  if (a[i] > m)
     <if (a[i] > m)
        m=a[i];>
}
write (m);
```

- Está certo?
- Os processos farão comparações duas vezes?

Resumo sobre sincronização

- Processos que lêem e escrevem em variáveis compartilhadas geralmente precisam de sincronização
- Ações atômicas podem ser "criadas" com < >
- Verificação dupla antes de atualizar é útil, principalmente quando a primeira verificação for falsa para boa parte dos processos (o segundo `if` não será executado)

Ações atômicas

Quais são as ações atômicas?

- Ação atômica realiza uma transformação de estado de forma indivisível
- Estados intermediários não podem ser vistos por outros processos
- Depende do hardware
  - Atribuições de variáveis são atômicas

Exemplo


```
int y = 0, z = 0;
co x = y+z;
// y = 1; z = 2;
oc
```

Considerações

- Tipos básicos (e.g. int) são armazenados em posições de memória que são lidas e escritas de forma atômica
- Manipulação de valores depende da cópia deles para registradores, operações nesses registradores e cópia de volta para a memória
- Cada processo tem seu próprio conjunto de registradores (mudança de contexto)
- Resultados intermediários resultantes de expressões complexas são armazenados em registradores ou na memória particular de cada processo

Sintaxe

- Precisamos de uma forma de definir expressões como atômicas
- Exemplo do algoritmo para encontrar o máximo
  - Usamos < e >

Ações atômicas e sincronização

- Para ações atômicas basta: `<` e `>`
- Para sincronização é necessário o `await`
  - `<await (B) S;>`
  - `B` é uma expressão booleana que especifica a condição de espera
  - `S` é uma sequência de comandos

Consequências ao se utilizar o `await`

- `<await (B) S;>`
  - `B` com certeza será verdade quando `S` for executado
  - Nenhum resultado intermediário de `S` será visto por outros processos
  - Se `B` é falso, só pode tornar-se verdade por ações de outros processos (uma expressão "protegida" por `await` pode esperar muito)

Formas de utilizar o `await`

- Exemplo: `<await (s > 0) s = s - 1;>`
- Para exclusão mútua basta: `<S;>`
- É uma ação atômica incondicional
  - `<x = x + 1; y = y + 1;>`

Formas de utilizar o `await`

- Para sincronização por condição basta: `<await (B);>`
  - `<await (count > 0);>`