MAC0438 - Programação Concorrente

Daniel Macêdo Batista

IME - USP, 22 de Março de 2013

Roteiro

Barreiras de sincronização

Melhorando a barreira em árvore

Barreiras simétricas

Barreiras de sincronização

Flags e coordenadores

- Primeira tentativa para zerar as variáveis

```
/* Worker */
arrive[i] = 1;
<await (continue[i] == 1);>
arrive[i] = 0;
```

```
/* Coordenador */
for [i = 1 to n] <await (arrive[i] == 1);>
for [i = 1 to n] continue[i] = 1;
for [i = 1 to n] continue[i] = 0;
```

- Problemas? Simulem

Flags e coordenadores

- Flag: Variável "erguida" por um processo para sinalizar que uma condição de sincronização é verdadeira
- **Princípios da sincronização por flags**
  - O processo que espera uma flag ser 1 é o processo que zera aquela flag
  - Uma flag não pode ter valor 1 antes da flag ter sido zerada
- Tentem corrigir o exemplo anterior

Flags e coordenadores


```
process Worker[i = 1 to n] {
   while (true) {
      codigo da tarefa i;
      arrive[i] = 1;
      <await (continue[i] == 1);>
      continue[i] = 0;
}

process Coordinator {
   while (true) {
      for [i = 1 to n] {
         <await (arrive[i] == 1);>
         arrive[i] = 0;
      }
      for [i = 1 to n] continue[i] = 1;
   }
}
```

Flags e coordenadores

- O algoritmo anterior exige um processo a mais que só coordena ("perde" um processador)
- O tempo de execução de cada iteração do coordenador é diretamente proporcional a `n`
  - Mesmo código para os Workers
  - Provavelmente todos os Workers vão chegar na barreira na mesma hora
- O ideal seria ter a verificação feita pelo coordenador em paralelo

Flags e coordenadores


```
process Coordinator {
   while (true) {
      for [i = 1 to n] {
         <await (arrive[i] == 1);>
         arrive[i] = 0;
      }
      for [i = 1 to n] continue[i] = 1;
   }
}
```

- Que parte do algoritmo pode ser paralelizada? (Onde podem ser criados mais processos?)

Flags e coordenadores


```
process Coordinator {
   while (true) {
      for [i = 1 to n] {
         <await (arrive[i] == 1);>
         arrive[i] = 0;
      }
      for [i = 1 to n] continue[i] = 1;
   }
}
```

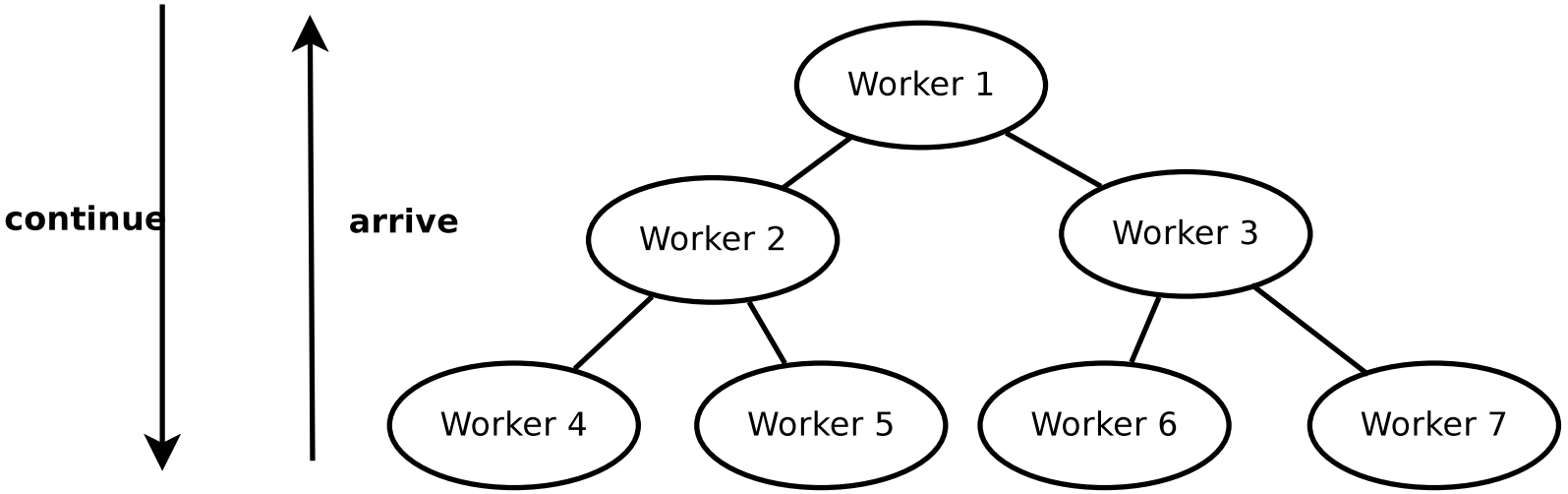
- `n` processos novos para os loops mas isso é custoso :(
- Mas já temos `n` processos!

Flags e coordenadores

- Ideia: usar os workers como coordenadores também
- Organizá-los em alguma estrutura que permita a comparação em paralelo ao invés de sequencial
- Sugestões?

Flags e coordenadores

- Barreira em árvore



- Proponham um algoritmo. Tem que ser diferente para folha, nó interno e raiz

Flags e coordenadores


```
folha L: arrive[L] = 1;
         <await (continue[L] == 1);>
         continue[L] = 0;

no interno I: <await (arrive[left] == 1);>
              arrive[left] = 0;
              <await (arrive[right] == 1);>
              arrive[right] = 0;
              arrive[I] = 1;
              <await (continue[I] == 1);>
              continue[I] = 0;
              continue[left] = 1; continue[right] = 1;

no raiz R: <await (arrive[left] == 1);>
           arrive[left] = 0;
           await (arrive[right] == 1);>
           arrive[right] = 0;
           continue[left] = 1; continue[right] = 1;
```

Flags e coordenadores

- No exemplo anterior, todos estão esperando o `continue` da raiz
- Como melhorar?

Flags e coordenadores

- O código pode ser melhorado se o processo raiz enviar um `continue` em broadcast
- Mas quem vai zerar o `continue`?

Melhorando a barreira em árvore

Com o `continue` em broadcast


```
folha L: if (round == 0) {
           arrive[L] = 1;
           <await (continue == 1);>
         }
         else {
           arrive[L] = 1;
           <await (continue == 0);>
         }

no interno I: ...

no raiz R: <await (arrive[left] == 1);>
           arrive[left] = 0;
           <await (arrive[right] == 1);>
           arrive[right] = 0;
           if (round == 0) { round = 1; continue = 1; }
           else { round = 0; continue = 0;}
```

- ☐ Resolve o problema? Simulem

Barreiras simétricas

Problema da barreira em árvore

- Os processos estão diferentes (nós interiores fazem mais coisa que as folhas e a raiz)
- O ideal seria ter uma solução que não diferenciasse os processos (processos diferentes – aumenta a chance de não chegarem na barreira na mesma hora)

Solução para os problemas da barreira em árvore

- **Barreiras simétricas**
- Não vai haver um processo especial esperando todos os outros chegarem na barreira
- Cada processo espera apenas outro (o que reduz o tempo de espera) – mas serão necessárias várias rodadas
- Primeiro vamos ver a solução com `n = 2`

Barreiras simétricas – 2 processos


```
/* barreira do processo W[i] */
<await (arrive[i] == 0);>
arrive[i] = 1;
<await (arrive[j] == 1);>
arrive[j] = 0;

/* barreira do processo W[j] */
<await (arrive [j] == 0);>
arrive[j] = 1;
<await (arrive[i] == 1);>
arrive [i] = 0;
```

Barreiras simétricas – 2 processos


```
/* barreira do processo W[i] */
<await (arrive[i] == 0);>
arrive[i] = 1;
<await (arrive[j] == 1);>
arrive[j] = 0;

/* barreira do processo W[j] */
<await (arrive [j] == 0);>
arrive[j] = 1;
<await (arrive[i] == 1);>
arrive [i] = 0;
```

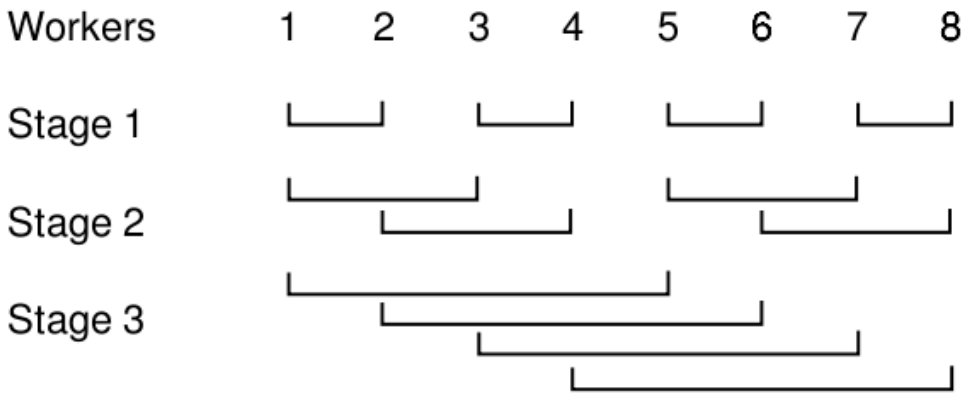
- A primeira linha de cada barreira é necessária?

Barreiras simétricas – qualquer `n`

- A ideia é tentar reproduzir o que foi feito na barreira em árvore
- O algoritmo vai ter várias rodadas e em cada rodada os processos fazem a barreira 2 a 2
- A diferença dos algoritmos está na definição de qual o processo com o qual cada um vai sincronizar

Barreira borboleta

- `Worker[1:n]` é um vetor de processos
- As barreiras são executadas entre os processos segundo o diagrama abaixo



- Rodada s
- Distância entre processos: 2^{s-1}

Barreira borboleta

- O nome vem por conta do diagrama de fluxo de dados borboleta do algoritmo FFT

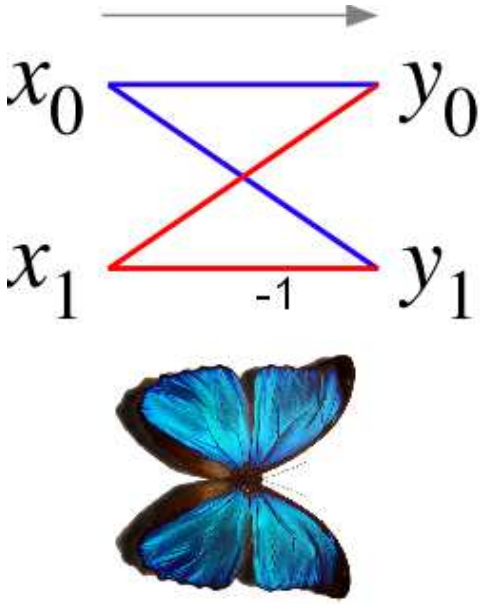


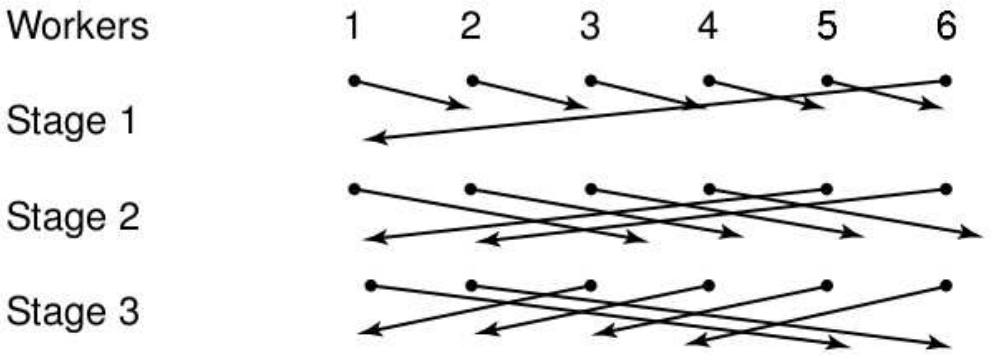
Figura 1: Fonte: Wikipedia

- No nosso caso, $x_0 = 1, x_1 = 4, y_0 = 2, y_1 = 3$

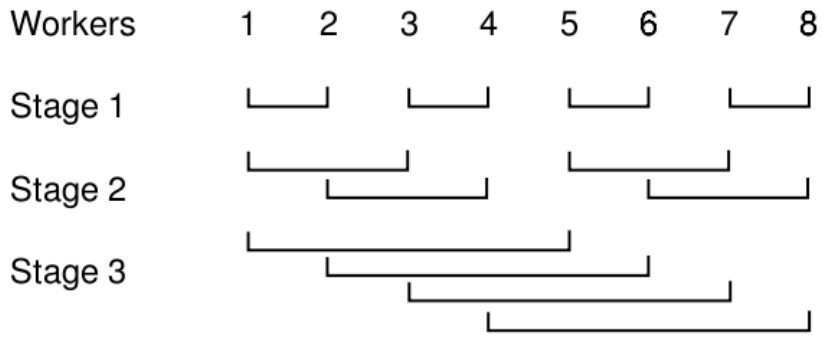
Barreira borboleta

- Após as $log_2(n)$ rodadas, todos os processos terão sincronizado entre si (direta ou indiretamente)
- Se n não for potência de 2?

Barreira de disseminação



Revisando os algoritmos das barreiras

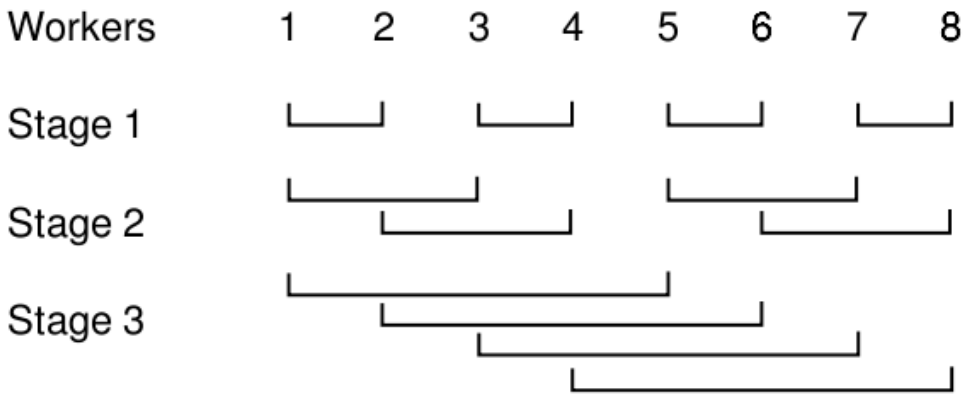



```
<await (arrive[i] == 0);>
arrive[i] = 1;
<await (arrive[j] == 1);>
arrive[j] = 0;

/* barreira do processo W[j] */
<await (arrive [j] == 0);>
arrive[j] = 1;
<await (arrive[i] == 1);>
arrive [i] = 0;
```

- Como saber a rodada atual? Simulem da forma que está para verificar se é necessário saber

Revisando os algoritmos das barreiras



- W[1] na rodada 1 faz `arrive[1] = 1`
- W[2] é lento e não viu ainda
- W[3] e W[4] sincronizam entre si e passam para a rodada 2
- W[3] está na rodada 2 e vê que `arrive[1] == 1`, mas o W[1] não fez isso para W[3]!!! Fez para W[2]!!!

Soluções para o problema de não saber os estágios

- Usar variáveis `arrive` para cada estágio (vai ser uma matriz)
- Outra solução? Como fazer o `arrive` com uma única dimensão representar o estágio atual?

Usando `arrive` como contador

- Tentem fazer o algoritmo

Usando `arrive` como contador


```
for [s = 1 to num_stages] {
  arrive[i]++;
  # determina o vizinho j para a rodada s
  while (arrive[j] < arrive [i]) skip;
}
```

- Simples assim :)