Implementação de Linguagens de Programação Lógica

Paralelismo

Ricardo Rocha
ricroc@ncc.up.pt
DCC-FCUP

Tópicos Avançados de Informática
Mestrado em Informática 2003/04

Porquê Paralelismo?

- **Eficiência das implementações sequenciais**
  - Os compiladores de Prolog baseados no modelo de execução sequencial WAM provaram ser altamente eficientes o que tornou o Prolog bastante popular.
- **Poder declarativo da linguagem**
  - Um programa é simplesmente um conjunto de cláusulas de predicados lógicos que podem ser vistos como especificações executáveis. As cláusulas são executadas pela ordem que são escritas e os objectivos de cada cláusula são executados da esquerda para a direita.
- **Potencial para a exploração implícita de paralelismo**
  - A execução paralela ocorre de modo automático por acção do próprio sistema de execução que detecta o paralelismo potencial do programa.
  - Solução mais geral e mais flexível que liberta o programador dos detalhes da execução paralela e que torna a programação paralela tão simples como a sequencial.

Principais Formas de Paralelismo

➤ Paralelismo-Ou

- ♦ Corresponde à execução simultânea do corpo de diferentes alternativas.
- ♦ Aparece quando mais do que uma cláusula define um predicado e uma chamada ao predicado unifica com a cabeça de mais do que uma cláusula.

```
a(X,Y) :- b(X), c(Y).
a(X,Y) :- d(X,Y), e(Y).
a(X,Y) :- f(X,Z), g(Z,Y).
```

➤ Paralelismo-E

- ♦ Corresponde à execução simultânea dos vários objectivos do corpo de uma cláusula.
- ♦ Aparece quando existe mais do que um objectivo presente numa questão ou no corpo de uma cláusula.

```
a(X,Y) :- b(X), c(Y).
```


Principais Formas de Paralelismo

➤ Paralelismo-E Independente

- ♦ Ocorre quando dois ou mais objectivos do corpo duma cláusula não partilham variáveis por instanciar.
- ♦ A potencial instanciação das variáveis de um objectivo é sempre compatível com o resultado dos outros objectivos.

```
a(X,Y) : - b(X), c(Y).
```

➤ Paralelismo-E Dependente

- ♦ Ocorre quando dois ou mais objectivos no corpo de uma cláusula têm variáveis comuns por instanciar.
- ♦ A execução simultânea desses objectivos pode levar a instanciações incompatíveis das variáveis comuns.

```
a(X,Y) : - d(X,Y), e(Y).
a(X,Y) : - f(X,Z), g(Z,Y).
```

Principais Formas de Paralelismo

➤ **Paralelismo-E Dependente**

- ♦ Uma solução é executar os objectivos independentemente até que ambos terminem. A compatibilidade das atribuições a variáveis comuns é verificada apenas no final (**back unification**). Esta solução é de certa forma semelhante ao Paralelismo-E Independente.

- ♦ Outra solução é executar os objectivos independentemente até que uma variável comum seja instanciada por um dos objectivos (designado por **produtor**). Os restantes objectivos (designados por **consumidores**) lêem a atribuição como sendo um argumento de entrada para a variável. A partir desse momento, sempre que o produtor encontrar uma nova atribuição para a variável comum, uma nova sequência pode ser iniciada por parte dos consumidores.

Sistemas Mais Conhecidos

➤ **Paralelismo-Ou**

- ♦ Aurora
- ♦ Muse

➤ **Paralelismo-E Independente**

- ♦ &-Prolog
- ♦ &ACE

➤ **Paralelismo-E Dependente**

- ♦ DASWAM
- ♦ ACE

Princípio da Ortogonalidade

- Intuitivamente, cada uma destas formas de paralelismo explora diferentes pontos de não-determinismo da linguagem. Sendo assim, deveria ser possível explorar todos eles simultaneamente num único sistema.
- Um tal sistema deverá verificar o **princípio da ortogonalidade**, ou seja, cada forma de paralelismo deverá ser explorada sem afectar a exploração de outra.
- Todavia, nenhum sistema eficiente foi ainda construído que suporte simultaneamente todas as formas de paralelismo. A aproximação com maior sucesso foi o sistema **Andorra-I** para a exploração simultânea de Paralelismo-Ou e Paralelismo-E Dependente.
- Um sistema que explore o máximo de paralelismo dos programas lógicos enquanto mantém a melhor performance possível, é o objectivo fundamental dos investigadores em programação lógica paralela.

Paralelismo-Ou: Motivação

Das formas de paralelismo, o Paralelismo-Ou é aquele que tem maior sucesso. A razão fundamental é que este parece ser mais fácil e mais produtivo de explorar.

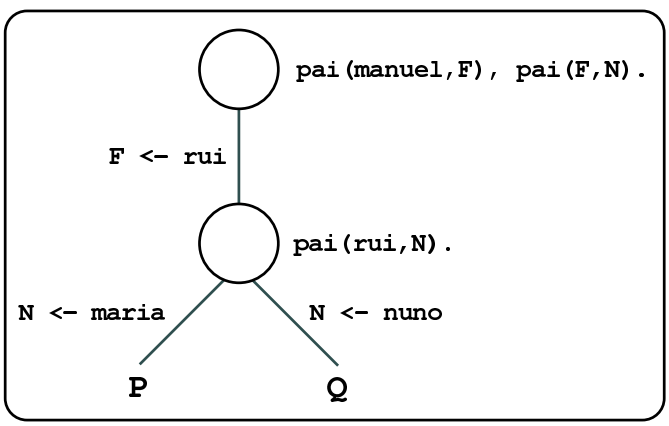
- **Simplicidade:** como as cláusulas de um mesmo predicado são independentes, é possível explorar Paralelismo-Ou sem solicitar qualquer anotação extra ao programador e sem necessitar de análises complexas em tempo de compilação.
- **Proximidade do modelo sequencial:** é possível explorar Paralelismo-Ou num modelo de execução bastante próximo do modelo sequencial. Isto significa que se torna mais fácil manter a semântica da linguagem, e que podemos tirar total vantagem da tecnologia existente de implementação sequencial para diminuir o custo do modelo paralelo sobre o modelo sequencial.
- **Granularidade:** para uma ampla classe de programas, o Paralelismo-Ou apresenta um potencial de paralelismo de granularidade alta. A granularidade de uma computação paralela diz respeito ao tamanho das tarefas que podem ser executadas sem interagir com as restantes tarefas processadas em simultâneo.

Paralelismo-Ou: Modelo Básico de Execução

- Inicialmente, todos os agentes encontram-se no estado de **suspensos**. Após a invocação de uma nova questão, um dos agentes inicia a execução.
- Sempre que um agente executa uma chamada que unifica com mais do que uma cláusula, ele cria um ponto de escolha **privado**. Pontos de escolha privados indiciam a presença de trabalho disponível para ser executado em paralelo.
- Logo que um agente suspenso detecta que existe trabalho disponível, ele solicita directamente a um dos agentes **ocupados** a partilha desse trabalho por forma a cooperar na sua exploração.
- Se um agente ocupado decide partilhar parte do seu trabalho, então ele torna **públicos** os pontos de escolha que pretende partilhar antes de sincronizar o seu estado com o agente suspenso.
- Assim que um agente explore todo o seu trabalho, este regressa ao estado de suspenso e reinicia a procura por novo trabalho disponível. O processo termina quando todo o trabalho tiver sido explorado e todos os agentes se encontrarem suspensos.

Paralelismo-Ou: Principais Problemas a Resolver

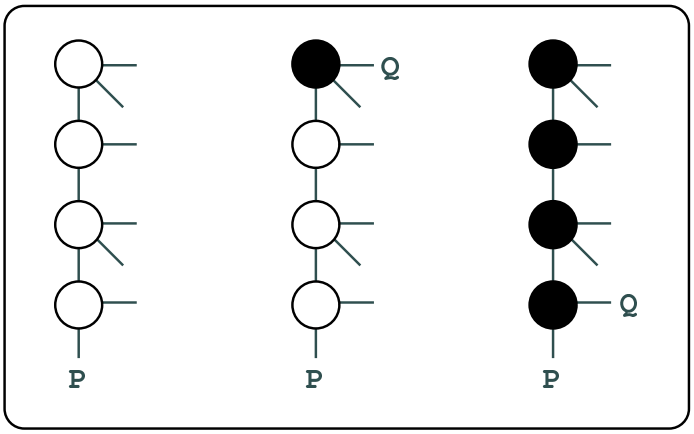
- **Múltiplos Ambientes**
  - **Problema:** como representar e aceder eficientemente aos múltiplos ambientes que coexistem simultaneamente?
  - **Solução:** utilizar mecanismos onde cada ambiente tem uma área privada onde guarda as suas atribuições condicionais. Uma atribuição a uma variável é condicional se a variável foi criada antes do ponto de escolha corrente.




```
pai(manuel, rui).
pai(pedro, joaquim).
pai(rui, maria).
pai(rui, nuno).
pai(maria, alberto).
```

Paralelismo-Ou: Principais Problemas a Resolver

- **Distribuição de Trabalho (Scheduling)**
  - **Problema:** como distribuir eficientemente o trabalho existente pelos agentes disponíveis de modo a minimizar o tempo total de execução?
  - **Solução:** existem duas políticas principais: distribuir trabalho o mais acima possível e distribuir trabalho o mais abaixo possível.



Paralelismo-Ou: Principais Modelos de Execução

Memória Partilhada:

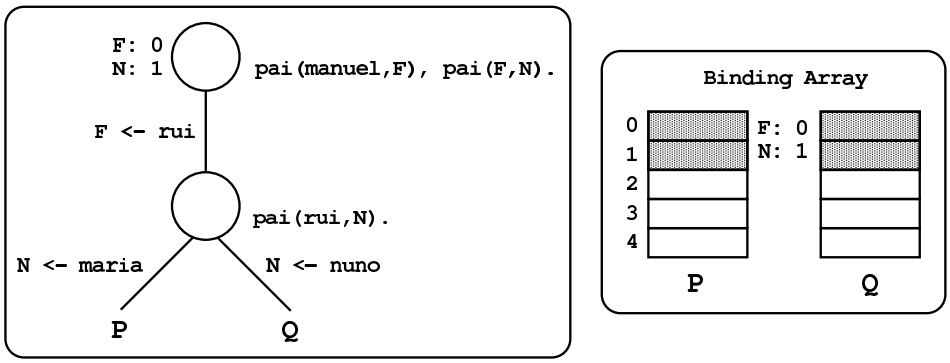
- **Binding Arrays**
- **Cópia de Ambientes**

Memória Distribuída:

- **Stack Splitting**

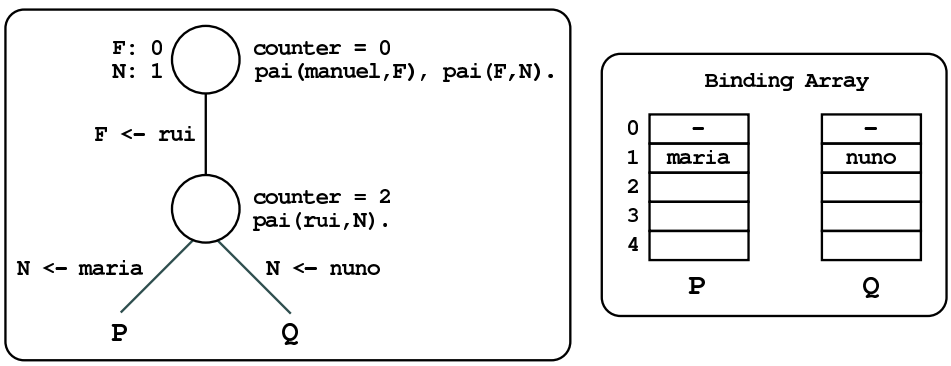
Binding Arrays

- Neste modelo, cada agente possui uma estrutura de dados auxiliar designada por **binding array**.
- Sempre que uma nova variável é criada esta é etiquetada com um **valor único** que identifica a sua posição no binding array. Isto é conseguido através de um **contador**. As variáveis são etiquetadas com o valor corrente do contador e em seguida este é incrementado. A numeração das variáveis ao longo de uma qualquer ramificação forma uma sequência estritamente crescente.



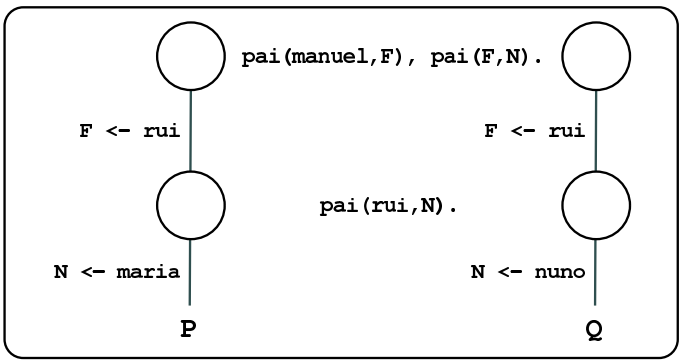
Binding Arrays

- Por forma a permitir que, sempre que uma nova alternativa for tomada num ponto de escolha, a sequência possa continuar, o valor do contador é guardado nos pontos de escolha.
- Uma atribuição a uma variável é **condicional** se o valor da sua etiqueta é menor que o valor do contador guardado no ponto de escolha corrente.
- O problema da coexistência de múltiplos ambientes é resolvido guardando as atribuições condicionais no binding array do próprio agente.



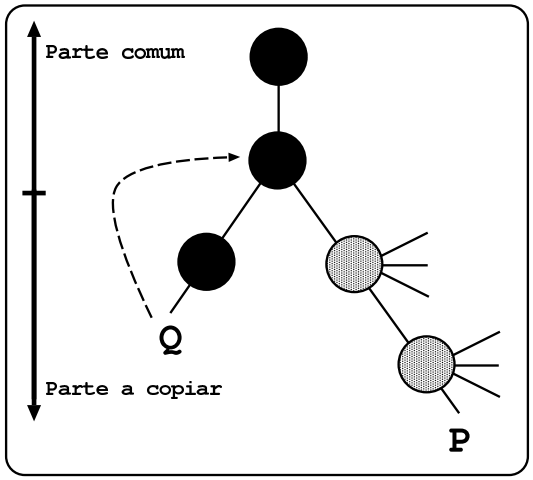
Cópia de Ambientes

- Neste modelo, cada agente mantém uma cópia separada do seu ambiente, no qual pode escrever sem causar conflito de atribuições.
- A partilha de trabalho é feita por **cópia das pilhas de execução** (pilha local, pilha de termos e trilha) do agente ocupado para o agente suspenso.
- Para maximizar a quantidade de trabalho partilhado e evitar que os agentes partilhem trabalho frequentemente (diminuindo assim o potencial número de operações de cópia), o **trabalho é distribuído o mais abaixo possível**.



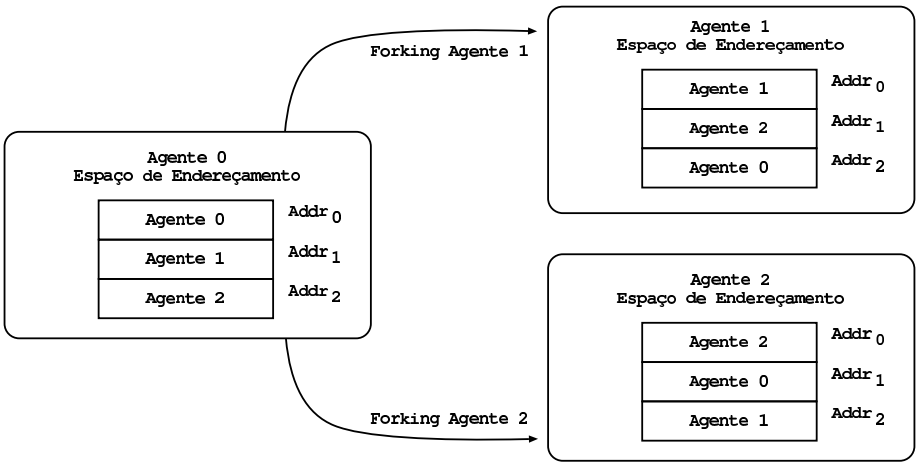
Cópia de Ambientes: Cópia Incremental

- **Problema:** a cópia de grandes quantidades de segmentos de memória pode degradar a performance do sistema.
- **Solução:** utilizar **cópia incremental**. A ideia fundamental da cópia incremental é baseada no facto de que parte das pilhas de execução pode ser comum a ambos os agentes, e por conseguinte não necessitam de ser copiadas.



Cópia de Ambientes: Mapeamento de Memória

- **Problema:** a cópia de segmentos de memória entre agentes requer realocação de endereços de modo a fazerem sentido no novo espaço de endereçamento.
- **Solução:** mapear a memória de tal modo que todos os agentes vejam as suas próprias áreas no mesmo endereço, ou seja, o espaço de endereçamento de cada agente, dum ponto de vista individual, tem início no mesmo endereço.



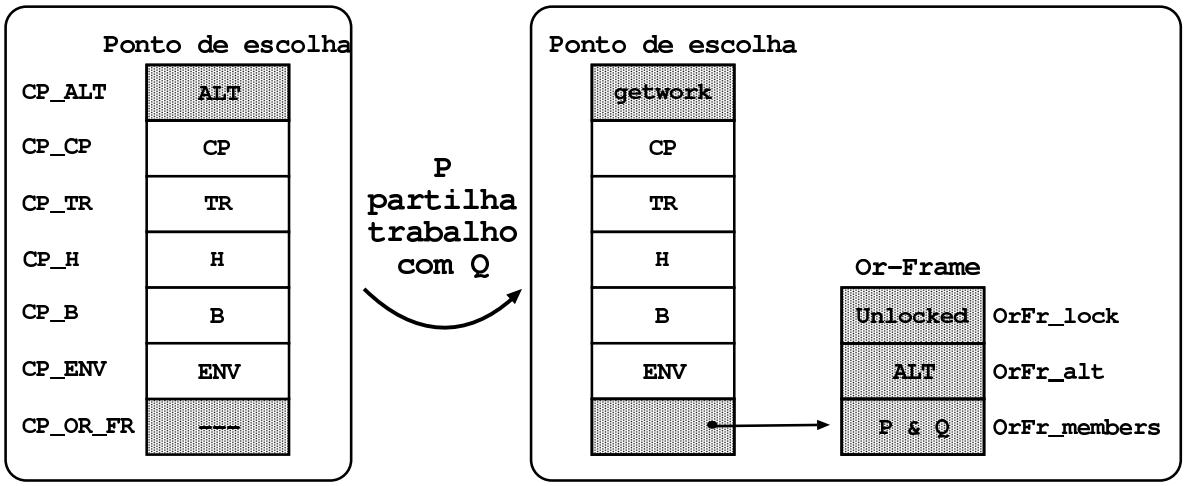
Cópia de Ambientes: Mapeamento de Memória

```
mapped_mem = mmap(NULL, n_agentes * local_space, ..., 0);
for (i = 0; i < n_agentes; i++)
  agente(i) = mapped_mem + local_space * i;
for (w = 1; w < n_agentes; w++)
  if (fork() == 0) {
    // libertar as regiões locais
    munmap(mapped_addr, local_space * n_agentes);
    // remapear as regiões locais
    for (i = 0; i < n_agentes; i++) {
      agente(i) = mapped_addr + local_space * ((n_agentes + i - w) % n_agentes);
      mmap(agente(i), local_space, ..., local_space * i);
    }
  }
```

- Este esquema permite operações de cópia bastante eficientes, pois evita a realocação de endereços.
- Suponha que o agente 2 quer copiar para o agente 1 um segmento de memória que começa no endereço **Addr** (do ponto de vista do agente 2). Então o endereço de destino deve ser **Addr + (agente(1) - agente(2))**.

Cópia de Ambientes: Sincronização

- **Problema:** como sincronizar o acesso aos pontos de escolha públicos de modo a garantir que cada alternativa é tomada uma e uma só vez?
- **Solução:** guardar as alternativas numa estrutura partilhada (**or-frame**) e usar uma pseudo-instrução (**getwork**) para sincronizar o acesso à estrutura.

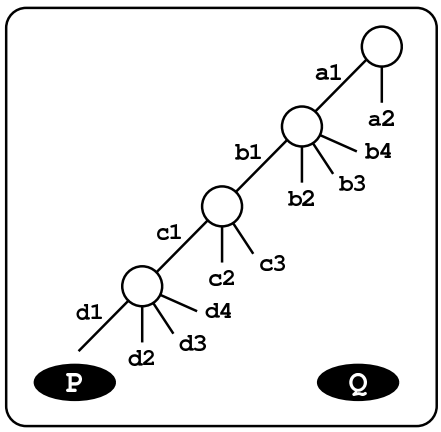


Stack Splitting

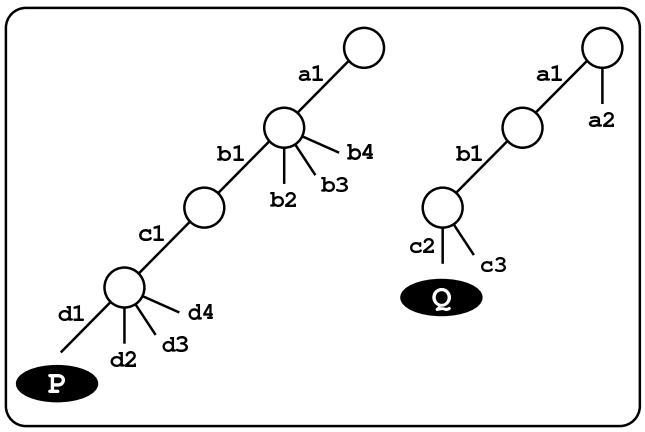
- **Problema:** utilizar mecanismos de exclusão mútua para sincronizar o acesso às or-frames em ambientes distribuídos pode levar a uma grande quantidade de troca de mensagens.
- **Solução:** dividir *à priori* as alternativas disponíveis pelos agentes envolvidos no processo de partilha de modo que cada agente fique desde logo com as suas alternativas. Esta técnica é designada por **stack splitting**.
- Neste modelo, a partilha de trabalho é igualmente feita por cópia das pilhas de execução acrescida duma fase posterior de **divisão de trabalho**.
- Existem três esquemas principais de divisão de trabalho:
  - Vertical Splitting
  - Horizontal Splitting
  - Diagonal Splitting

Stack Splitting: Vertical Splitting

➤ Os pontos de escolha são alternadamente divididos entre os dois agentes, começando pelo agente P que partilha trabalho.



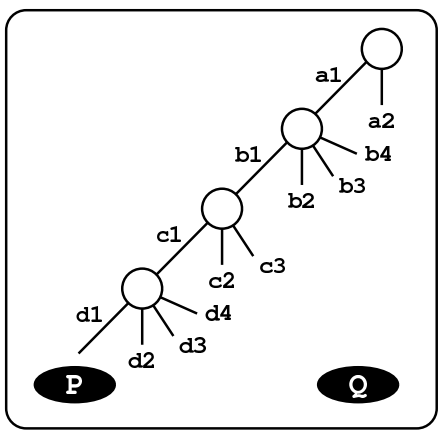
Antes



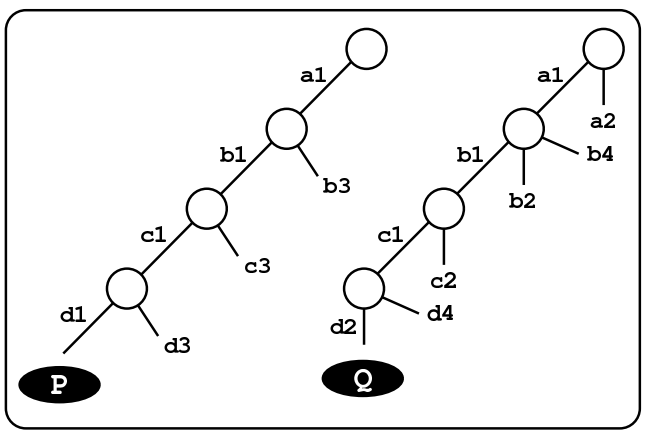
Depois

Stack Splitting: Horizontal Splitting

➤ As alternativas por explorar em cada ponto de escolha são alternadamente divididas entre o agente Q que pede trabalho e o agente P que partilha trabalho.



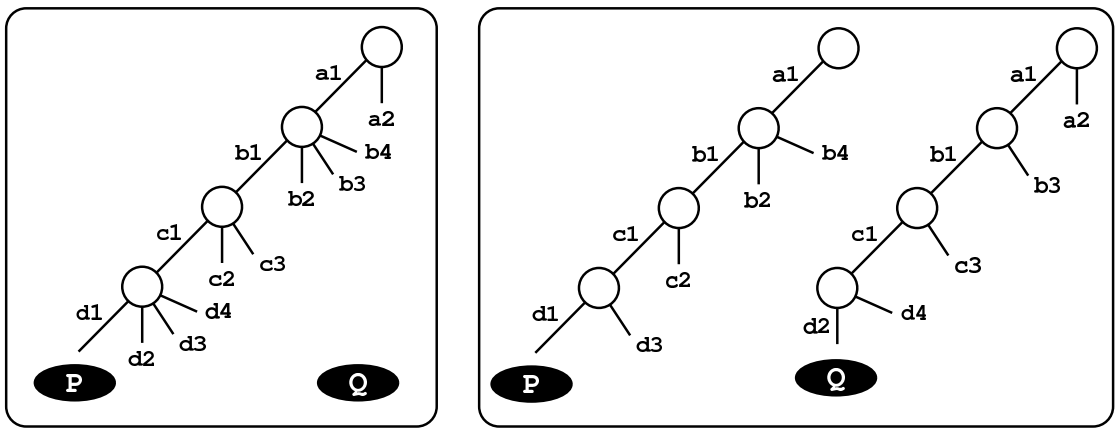
Antes



Depois

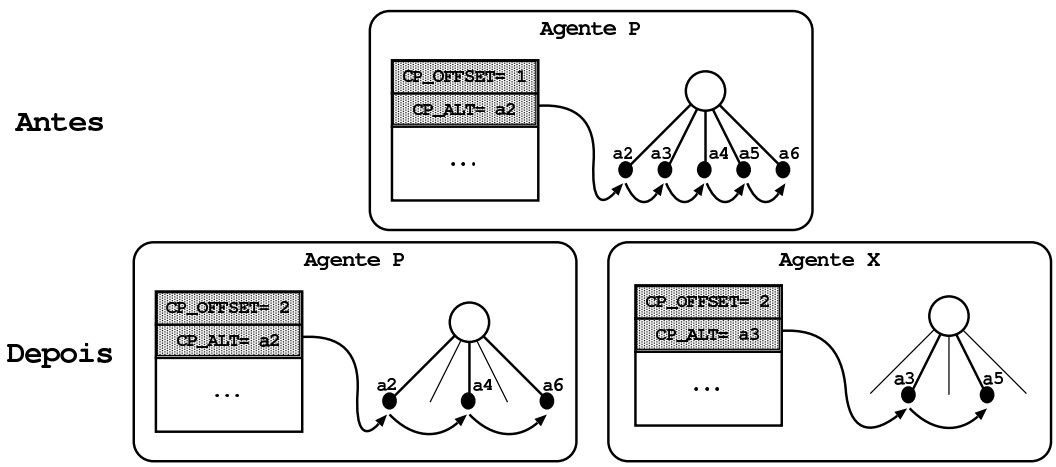
Stack Splitting: Diagonal Splitting

- O conjunto de todas as alternativas por explorar, independentemente dos pontos de escolha a que pertencem, é alternadamente dividido entre o agente Q que pede trabalho e o agente P que partilha trabalho.
- Este esquema é como que uma mistura dos dois anteriores e é o único que garante uma divisão equitativa das alternativas por explorar.



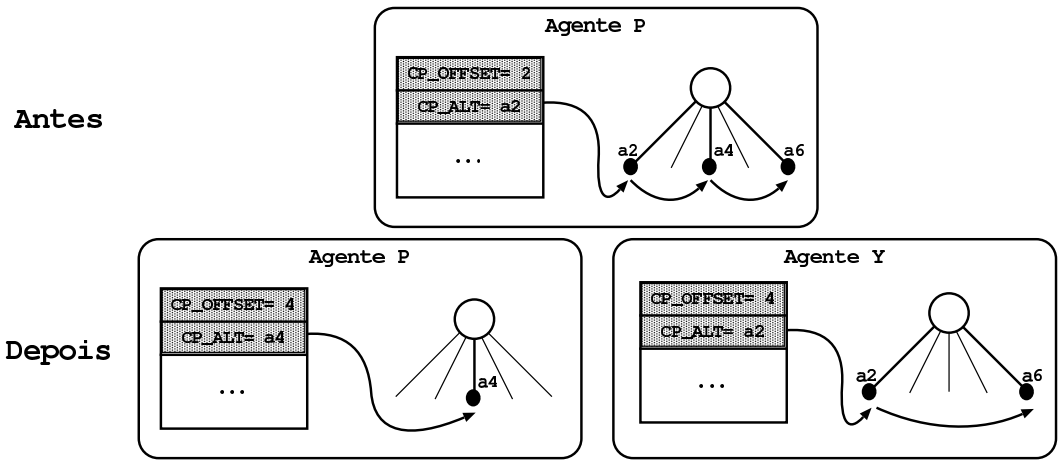
Stack Splitting: Como Dividir?

- Introduzir um novo campo nos pontos de escolha (**CP_OFFSET**) para indicar o deslocamento entre as alternativas que lhe pertencem.
- Para os pontos de escolha privados o valor do deslocamento é 1.
- Quando se partilha um ponto de escolha duplica-se o valor do deslocamento.



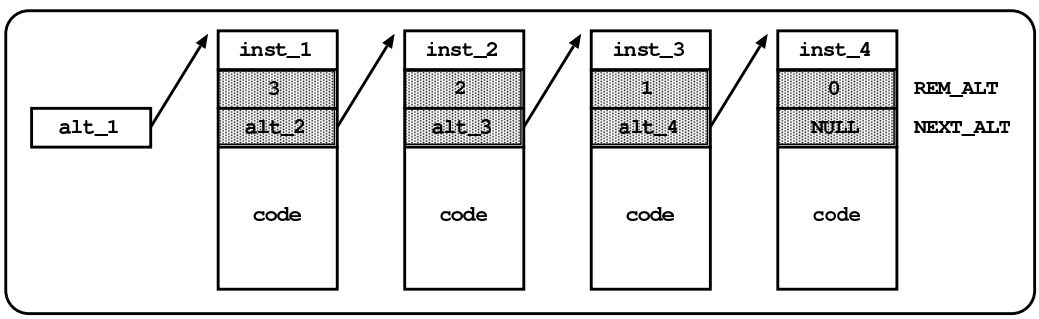
Stack Splitting: Como Dividir?

- O agente que não inicia o processo de divisão num dado ponto de escolha deve actualizar a referência para a próxima alternativa por explorar (campo **CP_ALT**).



Diagonal Splitting: Como Dividir?

- No esquema de Diagonal Splitting é necessário saber se o número de alternativas por explorar num ponto de escolha é par ou ímpar de modo a decidir qual o agente que inicia o processo de divisão no ponto de escolha seguinte.
- Uma possibilidade é percorrer a lista de alternativas por explorar e contar o seu número.
- Um esquema mais eficiente é incluir durante o processo de compilação informação relativa ao número de alternativas restantes a partir da alternativa corrente.



Diagonal Splitting: Como Dividir?

```
update_alternative(choice point CP) {
  offset = CP_OFFSET(CP)
  next_alt = CP_ALT(CP)
  if (offset > REM_ALT(next_alt)) next_alt = NULL
  else while (offset--) next_alt = NEXT_ALT(next_alt)
  CP_ALT(CP) = next_alt
}

diagonal_splitting(bottom choice point BCP, top choice point TCP) {
  if (requesting worker) starts = TRUE  // o agente que pede trabalho
  else starts = FALSE                   // inicia a divisão
  current_cp = BCP
  while (current_cp != TCP) {
    alt = CP_ALT(current_cp)
    if (alt != NULL) {
      if (starts == FALSE) update_alternative(current_cp)
      if ((REM_ALT(alt) / CP_OFFSET(current_cp)) mod 2 == 0)
        starts = !starts  // os agentes alternam
      CP_OFFSET(current_cp) *= 2
    }
    current_cp = CP_B(current_cp)  // CP_B refere o próximo ponto de escolha
  }
}
```