Algoritmos Distribuídos para Detecção de Predicados Globais*

L.M.A. Drummond [1] V.C. Barbosa [2]

RESUMO

Um dos aspectos mais importantes de depuradores de programas paralelos distribuídos consiste na facilidade de se estabelecerem *breakpoints* que possam ser descritos por predicados envolvendo estados globais e, então, denominados predicados globais.

Neste artigo consideramos o projeto de algoritmos distribuídos para a detecção de tais *breakpoints* em programas paralelos distribuídos e fornecemos quatro algoritmos, um para cada tipo diferente de *breakpoint*. Um dos algoritmos detecta a ocorrência de *breakpoints* incondicionais, enquanto os outros três detectam a ocorrência de *breakpoints* sobre predicados disjuntivos, predicados conjuntivos estáveis e predicados conjuntivos genéricos. Todos os algoritmos apresentados detectam os *breakpoints* nos estados globais mais adiantados em relação às propriedades envolvidas. No caso de *breakpoint* incondicional, tal estado global mais adiantado deve coincidir exatamente com os *breakpoints* incondiconais locais requisitados para os processos que realmente participam do *breakpoint*. No caso dos outros *breakpoints* (condicionais), detecta-se o estado global mais adiantado onde o predicado disjuntivo ou o conjuntivo considerado é verdadeiro.

ABSTRACT

The ability to set breakpoints, that can be expressed by predicates involving global states and thus called global predicates, stands as one of the most important issues in the debugging of message-passing programs.

We consider in this paper the design of fully distributed algorithms for the detection of such breakpoints in distributed parallel programs, and provide four algorithms, each for a different type of breakpoint. One of the algorithms detects the occurrence of unconditional breakpoints, while the other three detect the occurrence of breakpoints on disjunctive predicates, stable conjunctive predicates and generic conjunctive predicates. All the algorithms we present detect breakpoints in the form of earliest global states with respect to the particular property involved. In the case of unconditional breakpoint, such an earliest global state must coincide exactly with the requested local unconditional breakpoints for the processes that do actually participate in the breakpoint. In the case of the other (conditional) breakpoints, what is detected is the earliest global state at which either the disjunctive or conjunctive predicate under consideration is true.

[1]Departamento de Computação, Instituto de Matemática, UFF, R. São Paulo, s/n, 24040-110 Niterói-RJ;

[2]Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Caixa Postal 68511, 21945-970 Rio de Janeiro-RJ

* Este trabalho foi parcialmente financiado pelo CNPq e CAPES.

1 Introdução

Um aspecto fundamental na depuração de programas distribuídos consiste no estabelecimento de *breakpoints* expressos por condições que envolvem mais de um processo. A detecção de um predicado global, isto é, uma condição que depende do estado de múltiplos processos, constitui um problema importante em computação distribuída devido ao fato de os processos não terem acesso ao estado global em nenhum tempo, que é distribuído pelos diversos processos que compõem o sistema [9].

Este trabalho apresenta o projeto de algoritmos distribuídos para a detecção de predicados globais. Os predicados globais que consideramos podem ser de três tipos. O primeiro representa os *breakpoints* incondicionais, que são pontos pré-estabelecidos especificados distribuidamente entre os componentes do programa. No segundo tipo estão os predicados disjuntivos onde a condição global é dada pela disjunção lógica das condições locais (predicados locais) espalhadas pelo sistema. Finalmente, tratamos dos predicados conjuntivos que especificam uma condição global através de uma conjunção lógica de um conjunto de condições locais. No caso de predicados conjuntivos também consideramos a situação particular em que o *breakpoint* reflete um predicado estável, isto é, um predicado que uma vez verdadeiro permanece verdadeiro até o final da execução. Os algoritmos apresentados são capazes de informar os estados globais mais adiantados que satisfazem os predicados. As provas dos teoremas relativos às corretudes e complexidades destes algoritmos podem ser encontradas em [3].

Poucos autores tratam da detecção de predicados globais. Dentre os artigos mais significativos da área podemos citar o de Miller e Choi [10]. Eles propõem algoritmos para detecção de predicados disjuntivos e predicados do tipo *linked* (especificado por uma seqüência de eventos que podem ser ordenados pela relação "aconteceu antes"), sem qualquer compromisso com a detecção do estado global mais adiantado. O problema de detecção do estado global mais adiantado foi tratado por Manabe e Imase [8], que forneceram algoritmos para parar uma computação distribuída no estado global mais adiantado em que um predicado conjuntivo é satisfeito e exibir o valor de uma expressão calculada em um estado global em que um predicado disjuntivo é satisfeito. Estes algoritmos exigem que um histórico de execução seja gerado em execução prévia, o que aumenta os requerimentos de memória do algoritmo. Uma solução centralizada foi mais recentemente proposta por Garg e Waldecker [4] para detectar predicados conjuntivos. Outros trabalhos na área são incompletos ou tratam de questões relacionadas [2] [6] [11] [12] [13].

O restante deste artigo está organizado da seguinte forma: Na Seção 2 apresentamos o modelo básico do sistema distribuído, fazemos algumas considerações sobre a notação utilizada pelos algoritmos e os principais conceitos que envolvem o assunto. Na Seção 3, apresentamos o algoritmo DETECT_DP, usado para detectar *breakpoints* expressos como predicados disjuntivos. Nesta seção, também apresentamos, o algoritmo BROADCAST_WHEN_TRUE que detecta *breakpoints* expressos como predicados conjuntivos em aplicações que não trocam mensagens. O mesmo tem como objetivo principal simplificar a compreensão dos demais algoritmos que serão apresentados nas seções seguintes, já que estes outros algoritmos se utilizam das técnicas empregadas nos algoritmos DETECT_DP e BROADCAST_WHEN_TRUE. Na Seção 4, o algoritmo DETECT_UBP para a detecção de *breakpoints* incondicionais é apresentado. Os algoritmos para detecção de predicados conjuntivos estáveis e genéricos ,DETECT_STABLE_CP e DETECT_CP respectivamente, são apresentados na Seção 5. Finalmente, a Seção 6 conclui o artigo.

2 O Modelo

Neste trabalho, o programa a ser depurado é representado pelo grafo não direcionado $G = (N, E)$, com $n = |N|$ e $c = |E|$. Para $i = 1, \cdots, n$, $p_i \in N$ é um processo que pode se comunicar exclusivamente por troca de mensagens com os processos $p_j \in N$ tal que $(p_i, p_j) \in E$, onde $1 \leq j \leq n$. Os processos com os quais p_i pode se comunicar são chamados seus *vizinhos*. O conjunto de vizinhos de p_i é denotado por N_i. As arestas em E representam canais de comunicação bidirecionais que, conforme consideramos, são de capacidade infinita, a fim de que os processos nunca precisem parar ao tentarem enviar uma mensagem. Também consideramos que os processos são assíncronos, no sentido de que possuem relógios locais independentes e de que a entrega de mensagens entre vizinhos sofre atrasos finitos, mas imprevistos.

Uma abstração útil consiste em modelar a computação que acontece localmente em um processo como uma seqüência de eventos. Associados com um evento estão: a identificação do processo onde ele ocorre, o tempo local (dado pelo relógio local do processo) no qual o evento acontece, assim como as possíveis mudanças no estado local do processo ou o envio (recebimento) de mensagens para (de) vizinhos. Em um evento só se pode receber no máximo uma mensagem, embora nenhuma restrição exista em relação ao número de mensagens enviadas associadas a este evento. A computação distribuída que acontece sobre G é então naturalmente associada com o conjunto de eventos que ocorrem em todos os processos, e que denotaremos por V.

Apesar de o sistema ser totalmente assíncrono, existe uma relação de interdependência entre os diversos eventos que constituem a computação, a relação "aconteceu antes", definida por Lamport [7]. A relação "aconteceu antes", é usada na definição de "estados globais consistentes" ou mais sucintamente "estados globais" [1].

Um estado global, definido como uma partição (V_1, V_2) de V, pode ser visto como o conjunto de estados locais de cada processo (o estado no qual o processo foi deixado imediatamente depois da ocorrência de todos os eventos pertinentes de V_1) e um conjunto de mensagens para cada canal em cada direção (mensagens enviadas relacionadas com eventos em V_1 a serem recebidas em conexão com eventos em V_2). Entretanto, todos os *breakpoints* que examinamos são definidos exclusivamente em relação aos estados locais de alguns processos. Assim, um estado global pode ser considerado no nosso contexto como possuindo somente n estados locais. Além disso, consideramos que os tempos locais são incrementados cada vez que um evento ocorre e permanecem constantes entre eventos consecutivos. Desta forma, um estado local de um processo é determinado pelo tempo local do processo (tempos locais são na verdade contadores de eventos). Um estado global é então visto como um vetor de n componentes de tempos locais. Se φ é um estado global e $lt_i \geq 0$ denota o tempo local de p_i com que p_i participa de φ, para $p_i \in N$, então o componente de φ correspondente a p_i é $\varphi[i] = lt_i$.

Sob esta visão de um estado global, um vetor φ de n elementos com tempos locais é um estado global se e somente se não existir um $p_i \in N$ que receba uma mensagem em um tempo mais adiantado do que (ou em) $\varphi[i]$ que foi enviada por algum $p_j \in N_i$ mais tarde do que $\varphi[j]$. Sob este mesmo ponto de vista, podemos também afirmar que um estado global φ é um estado global mais adiantado para o qual uma certa propriedade é satisfeita se e somente se não existir nenhum outro estado global φ', no qual a propriedade é satisfeita, tal que $\varphi'[k] \leq \varphi[k]$ para todo $p_k \in N$.

Algoritmos para resolver o problema de detecção de *breakpoint* não devem nunca perder um estado global que satisfaça a condição estabelecida.

Freqüentemente, existe ainda a necessidade de que o estado global detectado seja o primeiro no qual o predicado é satisfeito, ou seja, deseja-se obter o estado global mais adiantado no qual uma certa propriedade é satisfeita.

Os algoritmos distribuídos para detecção de *breakpoints* são avaliados em relação ao *overhead* decorrente da monitoração da computação. Da mesma forma que as medidas de complexidade normais adotadas pelos algoritmos distribuídos assíncronos, este *overhead* é dado como expressões do pior caso pelo número de *bits* de mensagens adicionais transmitidas entre vizinhos e o tempo adicional para execução. Uma solução equivalente à contagem de *bits* das mensagens é a contagem do número de mensagens. Neste trabalho, entretanto, a comunicação adicional é muitas vezes decorrente de campos adicionais ligados às mensagens da própria computação, e isto não seria visível se adotássemos contadores de mensagens.

Enquanto a contagem do número de *bits* das mensagens a fim de se obter a complexidade de mensagens é quase sempre um problema simples, a definição de complexidade de tempo requer um pouco mais de elaboração devido ao assincronismo inerente ao sistema. Normalmente, considera-se que a computação local não gasta tempo, e então a complexidade de tempo expressa a cadeia mais longa do tipo "receber uma mensagem e enviar uma mensagem como conseqüência" ocorrendo durante a computação. Entretanto, tal cadeia causal é entremeada com a ocorrência de eventos não relacionados às mensagens, que algumas vezes podem ser significativos, e assim a hipótese de que computação local não gasta tempo pode se tornar errada. A convenção usada neste trabalho é expressar a complexidade de tempo como um par de medidas, uma sendo a tradicional medida, à qual eventos não relacionados às mensagens não contribuem (chamada de complexidade de tempo global para evidenciar sua dependência do sistema de um modo geral), e uma outra para estimar, dentro de um único processo, as cadeias causais envolvendo também eventos não relacionados às mensagens (chamada de complexidade de tempo local).

Muitas vezes, precisaremos nos referir à complexidade de mensagens da computação propriamente, definida como $O(c,(n,e))$ mensagens, ou mais sucintamente $O(c)$ mensagens. Do mesmo modo, nós consideraremos que todo relógio local de processo só pode representar tempos até um valor T, isto é, $lt_i \leq T$ para todo $p_i \in N$. T pode ser arbitrariamente grande e pode ser referenciado nas complexidades dos algoritmos.

3 Algoritmos Preliminares

Em nossos algoritmos, associado a cada processo p_i temos um outro processo q_i. Processos $q_1, \cdots, q_n$ se comunicam por meio de mensagens enviadas sobre os canais de comunicação correspondentes às arestas em E também, e são favorecidos com as seguintes habilidades para $1 \leq i \leq n$.

- Toda mensagem recebida ou enviada por p_i é interceptada por q_i, que pode anexar novos campos à mensagem ou retirar alguns campos da mensagem antes de transmitila para p_i ou enviá-la para algum vizinho de p_i.
- O processo q_i é ativado (e então p_i é suspenso, enquanto lt_i permanece constante) quando lt_i se torna igual a lub_i (no caso de *breakpoints* incondicionais) ou na mudança no valor do predicado local lp_i (no caso de *breakpoints* condicionais), ou ainda no envio de uma mensagem por p_i ou o recebimento de uma mensagem de q_j tal que $p_j \in N_i$.
- O processo q_i pode suspender a execução de p_i a qualquer tempo, e também continuá-la (possivelmente depois de ter alterado o seu contexto).

Cada par de processos p_i e q_i compartilha um único processador, que é comutado entre os dois para execução. O processo p_i executa somente quando q_i não está executando,

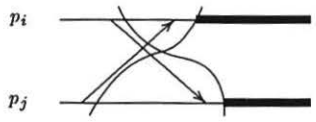

Figure 1: Dois estados globais mais adiantados onde o predicado disjuntivo é satisfeito

exceto se este for suspenso por q_i. Por simplicidade, quando necessário nos referimos a tal par de processos como um nó.

Os algoritmos para detecção de *breakpoints* são baseados na seguinte solução geral. Para $1 \leq i \leq n$, os processos q_i mantêm um vetor gs_i de comprimento n representando o estado global a ser detectado. Este vetor é inicializado com zeros (representando o estado global mais adiantado da computação) e é atualizado quando o nó recebe informação de outros nós. Tal informação é enviada de nó para nó ou por meio de mensagens especiais, chamadas *broadcast*, ou como campos adicionais ligados às mensagens do tipo *computação*, que são as próprias mensagens da aplicação. Esta informação, quando enviada por q_i, inclui o vetor gs_i, e pode englobar outros campos adicionais, dependendo do tipo de *breakpoint* a ser detectado. Um nó, ao receber esta informação, pode detectar a satisfação do *breakpoint*.

3.1 Detecção de Predicados Disjuntivos

O algoritmo DETECT_DP detecta, como já foi dito, *breakpoints* sobre predicados disjuntivos. Tal *breakpoint* é um estado global no qual pelo menos um dos predicados locais dos processos que participam do *breakpoint* é satisfeito. O estado global mais adiantado no qual um predicado disjuntivo é satisfeito não tem que ser único. Assim, é possível que mais de um processo detecte a ocorrência do *breakpoint*, entretanto em estados globais diferentes.

A principal contribuição do algoritmo DETECT_DP em relação aos trabalhos anteriores [8] [10] reside nos fatos deste detectar o estado global mais adiantado que satisfaz o predicado e não requisitar a gravação prévia de um histórico de execução.

Na Figura 1, como nas demais ilustrações deste trabalho, os processos são representados por eixos de tempos locais, nos quais os tempos locais crescem da esquerda para a direita, as setas representam as mensagens, um estado global é um corte (linha) dividindo os eixos e as barras mais grossas representam os períodos durante os quais os predicados locais são verdadeiros. Assim, na Figura 1 existem claramente dois estados globais mais adiantados nos quais pelo menos um dos predicados locais, lp_i ou lp_j, é **true**.

O algoritmo distribuído DETECT_DP é relativamente simples. Ele não emprega mensagens do tipo *broadcast* e anexa às mensagens de *computação*, enviadas por q_i tal que $p_i \in N$, o vetor $gs_i(lt_i)$, além de um *bit* de *status* (a ser discutido mais adiante). Este vetor é idêntico ao gs_i em todos os componentes exceto o i-ésimo, que é dado por lt_i. O valor de lt_i neste caso corresponde ao tempo local em p_i quando este enviou a mensagem que q_i interceptou, ou seja, reflete o estado local de p_i imediatamente depois de enviar a mensagem. As mensagens de *computação* enviadas por q_i são então triplas como ("status bit", $gs_i(lt_i)$, *body*), onde *body* se refere ao conteúdo da mensagem que q_i recebeu de p_i. Da mesma forma, quando q_i recebe uma mensagem ("status bit", gs_j, *body*) de q_j tal que $p_j \in N_i$, é a parte da mensagem *body* que é encaminhada a p_i, a fim de que os processos em N só fiquem envolvidos com as mensagens de computação propriamente. A essência do algoritmo DETECT_DP é a seguinte para $p_i \in N$. A variável lp_i é inicializada com **false**, e consideramos que ela nunca se torna **true** se p_i não participa do *breakpoint*. Sempre que q_i detecta que lp_i se tornou **true**, ele atribui a $gs_i[i]$ o valor lt_i e declara detectado

o *breakpoint* sobre o predicado disjuntivo no estado global gs_i. Como toda mensagem recebida de q_j tal que $p_j \in N_i$ antes de lt_i carregava uma cópia da visão de q_j do estado global com o j-ésimo componente atualizado com o tempo em que a mensagem foi enviada, gs_i é realmente um estado global. Para assegurar que este seja também o estado global mais adiantado em relação ao predicado disjuntivo, é utilizado o "status bit". Este *bit* indica, ao ser recebido junto à mensagem de *computação*, se qualquer outro estado global já foi detectado. O algoritmo DETECT_DP é constituído por um conjunto de ações a ser executado por q_i. Duas variáveis adicionais empregadas pelo algoritmo são: $found_i$ e $found\_elsewhere_i$, ambas inicializadas com **false** e indicando, respectivamente, se q_i detectou o *breakpoint* sobre o predicado disjuntivo e se tal predicado já foi detectado em algum outro nó , em cujo caso, q_i não teria detectado o estado mais adiantado.

A seguir é apresentado o algoritmo para detecção de predicados disjuntivos, que apresenta complexidade de mensagens de $O(cn\log T)$ *bits*, complexidade de tempo global de $O(1)$, complexidade de tempo local de $O(n)$ por mensagem recebida, e requer $O(n\log T)$ *bits* de memória por processo.

Ações em q_i para o algoritmo DETECT_DP:

(1) Ao detectar que lp_i se tornou **true**:

```
if not (found_i or found_elsewhere_i) then
  begin
    gs_i[i] := lt_i;
    found_i := true
  end;
```

(2) Ao receber $(body)$ de p_i destinado a $p_j \in N_i$:

Encaminhe $(found_i$ **or** $found\_elsewhere_i, gs_i(lt_i), body)$ para q_j;

(3) Ao receber $(bit_j, gs_j, body)$ de q_j tal que $p_j \in N_i$:

```
found_elsewhere_i := found_elsewhere_i or bit_j;
if not (found_i or found_elsewhere_i) then
  for k := 1 to n do
    if gs_i[k] < gs_j[k] then
      gs_i[k] := gs_j[k];
Encaminhe (body) para p_i;
```

3.2 Propagação de Informação Global

A detecção dos outros tipos de *breakpoints* considerados neste trabalho é uma tarefa mais difícil em comparação com a detecção de *breakpoints* sobre predicados disjuntivos. Estes outros casos requerem que se monitore uma informação global. A propagação desta informação global faz uso de mensagens de *broadcast* que apresentamos anteriormente.

Em geral, o processo q_i tal que $p_i \in N$ além do vetor gs_i mantém um outro vetor de variáveis lógicas com sua visão local da condição global a ser monitorada e detectada. Quando disseminado por q_i, este vetor é também acompanhado por gs_i, a fim de que sempre que q_i detecte localmente que a condição global ocorreu (examinando seu vetor), este possa associar os conteúdos de gs_i com o estado global no qual a condição ocorreu.

Mensagens de *broadcast* são enviadas por q_i desde que p_i seja um dos processos que participe na condição global a ser detectada e ou seu *breakpoint* incondicional seja atingido ou seu predicado local se torne verdadeiro. O *broadcast* empregado é do tipo "por enchente", isto é, a informação é enviada por q_i a todo q_j tal que $p_j \in N_i$, e assim por diante até chegar a todos os nós. Durante esta propagação de informação, um vetor gs_j de algum

q_j tal que $p_j \in N_i$ é usado por q_i para atualizar gs_i. Além disso, gs_j e o outro vetor que o acompanha são usados para atualizar a visão local em q_i da condição global sendo monitorada.

Certamente, alguns cuidados são necessários com este simples procedimento de propagação, tal como nunca enviar a um processo a mesma informação que foi recebida deste, a fim de garantir que a propagação termine. Além disso, foi adotada uma regra "encaminhe quando verdade" para a propagação da informação em alguns dos algoritmos. Por esta regra, um nó participa no *broadcast* (isto é, encaminha a informação que ele recebe) somente quando sua condição local é satisfeita. Claramente, se nenhuma mensagem fosse enviada durante a computação, então este *broadcast* seria suficiente para a detecção do tipo desejado de *breakpoint*. Em tal caso, todo nó que tivesse um vetor com valores **true** para todos os processos da computação declararia a detecção do *breakpoint* no estado global fornecido pelo vetor de estado global.

O algoritmo BROADCAST_WHEN_TRUE faz esta detecção na ausência de mensagens relacionadas à computação, desde que a condição global sob monitoração seja estável. Neste algoritmo, o processo q_i mantém uma variável lógica lc_i para indicar se a condição local com a qual p_i participa (se for o caso) na condição global a ser detectada é satisfeita. Esta é inicializada com **false** se p_i realmente participa na condição global, ou com **true** caso contrário. A estabilidade significa que para todo $p_k \in N$ lc_k nunca se torna **false** uma vez que já tenha se tornado **true**. O vetor associado com a visão de q_i da condição global é chamado de gc_i. Para $1 \leq k \leq n$, $gc_i[k]$ é inicializado com o mesmo valor atribuído inicialmente a lc_k. Somente mensagens de *broadcast* são empregadas no algoritmo (já que a própria computação não envia mensagens), constituídas pelo par (gc_i, gs_i), quando q_i é o processo que as envia. Como no caso do algoritmo DETECT_DP discutido anteriormente, uma variável lógica $found_i$, inicializada com **false**, é empregada para indicar se q_i detectou a ocorrência da condição global. Além disso, uma outra variável lógica, $changed_i$, é usada por q_i para assegurar que uma mensagem de *broadcast* nunca seja enviada para um nó se esta não for diferente da última que foi enviada para aquele mesmo nó.

Apresentamos a seguir o algoritmo BROADCAST_WHEN_TRUE.

Ações em q_i para o algoritmo BROADCAST_WHEN_TRUE:

(1) Ao detectar que lc_i se tornou **true**:

```
gc_i[i] := lc_i;
gs_i[i] := lt_i;
if gc_i[1] ∧ ··· ∧ gc_i[n] then
  found_i := true
else
  Envie (gc_i, gs_i) para todo q_k tal que p_k ∈ N_i;
```

(2) Ao receber (gc_j, gs_j) de q_j tal que $p_j \in N_i$:

```
if not found_i then
  begin
    changed_i := false;
    for k := 1 to n do
      if gs_i[k] < gs_j[k] then
        begin
          gs_i[k] := gs_j[k];
          gc_i[k] := gc_j[k];
          changed_i := true
        end;
    if lc_i and changed_i then
      if gc_i[1] ∧ ··· ∧ gc_i[n] then
```

$found_i$:= **true**
else
Envie (gc_i, gs_i) para todo q_k tal que $p_k \in N_i$
end;

O algoritmo BROADCAST_WHEN_TRUE possui complexidade de mensagens de $O(n^2 \log T)$ *bits*, complexidade de tempo global de $O(n)$, complexidade de tempo local de $O(n)$ por mensagem recebida, e requer $O(n \log T)$ *bits* de memória por processo.

4 Detecção de Breakpoints Incondicionais

Nesta seção apresentamos DETECT_UBP, um algoritmo distribuído para detectar a ocorrência em uma computação distribuída de um *breakpoint* incondicional. Este *breakpoint* incondicional é especificado como um tempo local representado por lub_i para $p_i \in N$, para cada processo que realmente participa do *breakpoint*. Para processos p_i que não participam no *breakpoint*, adota-se $lub_i = \infty$, a fim de que lt_i nunca seja igual a lub_i.

Não existem na literatura algoritmos que realizem este tipo de detecção.

O algoritmo DETECT_UBP pode ser considerado uma combinação dos algoritmos DETECT_DP e BROADCAST_WHEN_TRUE, discutidos no seção anterior, pois a detecção de *breakpoints* incondicionais requer a detecção de uma condição global, como no algoritmo BROADCAST_WHEN_TRUE, mas também requer um tratamento de mensagens de *computação*, da mesma forma que no algoritmo DETECT_DP.

As variáveis empregadas são essencialmente as mesmas que as empregadas nos algoritmos do seção anterior, com apenas algumas diferenças. Para q_i tal que $p_i \in N$, a variável lc_i é substituída pela igualdade $lt_i = lub_i$, e em lugar do vetor gc_i usa-se o vetor ub_i. Cada elemento do vetor ub_i pode assumir os valores **false**, **true** ou **undefined** . Os valores **true** e **false** se referem aos processos que participam do *breakpoint* e atingiram ou não, respectivamente, o seu *breakpoint* incondicional local, enquanto o valor **undefined** se refere aos processos que não participam do *breakpoint* incondicional. Assim, os elementos do vetor ub_i são inicializados com **false** para as posições referentes aos processos que participam do *breakpoint* e com **undefined** para os processos que não participam.

O algoritmo DETECT_UBP funciona da seguinte forma. Quando q_i detecta que $lt_i = lub_i$, os elementos $ub_i[i]$ e $gs_i[i]$ são atualizados e é iniciado um *broadcast* para difundir os vetores ub_i e gs_i. Como já visto no algoritmo BROADCAST_WHEN_TRUE, um nó não prossegue com o *broadcast* caso não tenha ainda atingido o seu *breakpoint* local, exceto quando este nó não faz parte do *breakpoint* incondicional . Além disso, mensagens de *broadcast* duplicadas nunca são enviadas por nenhum nó. Em relação às mensagens de *computação*, estas recebem o mesmo tratamento do algoritmo DETECT_DP, ou seja, são sempre enviadas com os vetores ub_i e $gs_i(lt_i)$ anexados. A detecção do *breakpoint* incondicional ocorre para q_i quando é verificado que $ub_i[k] \neq$ **false** para todo $p_k \in N$, isto é, todo processo ou atingiu seu *breakpoint* incondicional local ou não está participando do *breakpoint*.

A principal dificuldade no projeto do algoritmo DETECT_UBP está na detecção de erros. Estes erros podem ocorrer quando o conjunto de *breakpoints* incondicionais locais não corresponde a um estado global. Para detectar esta situação procedemos da seguinte forma. Uma variável lógica in_error_i, inicializada com **false**, é empregada por q_i para indicar se uma condição de erro foi detectada. Quando q_i recebe uma mensagem de *computação*, com os vetores ub_j e gs_j anexados, de algum q_j tal que $p_j \in N_i$, se $ub_j[j] =$ **true** e $ub_i[i] =$ **false**, então um erro ocorreu na determinação do *breakpoint* incondicional, pois

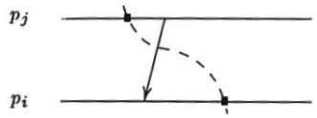

Figure 2: Erro na colocação dos *breakpoints* incondicionais locais

p_i nunca atingirá seu *breakpoint* incondicional local de tal forma que este seja consistente com o *breakpoint* incondicional local de p_j do ponto de vista de um estado global. Esta situação é ilustrada na Figura 2, na qual a partição do conjunto de eventos indicada pela linha tracejada não constitui um estado global.

O tratamento de erros se torna um pouco mais complicado devido à possibilidade de se ter nós para os quais nenhum *breakpoint* local incondicional é especificado. Se uma cadeia causal de mensagens de *computação*, iniciando em q_ℓ tal que $ub_\ell[\ell]$ = **true** e passando por um conjunto de nós q_k para os quais $ub_k[k]$ = **undefined**, levar a q_i tal que $ub_i[i]$ = **false**, então um erro deve ser detectado exatamente como no caso discutido anteriormente. Para resolver isto, deve-se ir atribuindo **true** a $ub_k[k]$ para todos os q_k's no recebimento de mensagens de *computação* de processos que tenham associado ao *breakpoint* incondicional local **true**.

Os nós que não participam do *breakpoint* incondicional também complicam a detecção do estado global mais adiantado. Cadeias causais de mensagens de *computação* partindo de q_ℓ tal que $ub_\ell[\ell]$ = **undefined** podem levar a estados globais distintos, dependendo de chegarem a q_i tal que $ub_i[i]$ = **false** ou $ub_i[i]$ = **true**, como ilustrado na Figura 3, cujas partes (a) e (b) descrevem, respectivamente, os dois casos. Somente no primeiro caso q_i deveria atualizar gs_i de acordo com os vetores anexados na mensagem de *computação* recebida, mas o processo que envia a mensagem não tem como saber disso. A estratégia utilizada para tratar este problema é a seguinte. Se $ub_i[i]$ = **undefined**, q_i, além de manter gs_i como uma visão local do estado global a ser detectado, também mantém uma visão alternativa, representada por alt_gs_i, que é inicializado como gs_i, atualizado no recebimento de mensagens de *computação* e *broadcast* e anexado às mensagens de *computação* enviadas por q_i, enquanto gs_i é apenas atualizado quando são recebidas mensagens de *broadcast*.

Assim, quando q_i recebe mensagens de *computação*, gs_i ou alt_gs_i podem ser atualizados se, respectivamente, $ub_i[i]$ = **false** ou $ub_i[i]$ = **undefined**. Deste modo, no caso de $ub_i[i]$ = **undefined**, como permitimos atualizações em alt_gs_i que não são realizadas em gs_i, temos que $gs_i[k] \leq alt\_gs_i[k]$ para todo $p_k \in N$, e por isso gs_i pode ser um estado global mais adiantado do que alt_gs_i.

A seguir são apresentadas as ações que compõem o algoritmo DETECT_UBP.

Ações em q_i para o algoritmo DETECT_UBP:

(1) Ao detectar que $lt_i = lub_i$:

```
if not in_error_i then
  begin
    ub_i[i] := true;
    gs_i[i] := lt_i;
    if ub_i[k] ≠ false for all k = 1, ..., n then
      found_i := true
    else
      Envie (broadcast, ub_i, gs_i, nil) para todo q_k tal que p_k ∈ N_i
  end;
```

(2) Ao receber $(broadcast, ub_j, gs_j, \text{nil})$ de q_j tal que $p_j \in N_i$:

```
if not (in_error_i or found_i) then
  begin
    changed_i := false;
    for k := 1 to n do
      if gs_i[k] < gs_j[k] then
        begin
          gs_i[k] := gs_j[k];
          ub_i[k] := ub_j[k];
          changed_i := true
        end;
    if ub_i[i] = undefined then
      for k := 1 to n do
        if alt_gs_i[k] < gs_j[k] then
          alt_gs_i[k] := gs_j[k];
    if (lub_i ≠ false) and changed_i then
      if ub_i[k] ≠ false para todo k = 1, ..., n then
        found_i := true
      else
        Envie (broadcast, ub_i, gs_i, nil) para todo q_k tal que p_k ∈ N_i
  end;
```

(3) Ao receber $(body)$ de p_i destinado a $p_j \in N_i$:

```
if ub_i[i] = undefined then
  Encaminhe (computation, ub_i, alt_gs_i(lt_i), body) para q_j
else
  Encaminhe (computation, ub_i, gs_i(lt_i), body) para q_j;
```

(4) Ao receber $(computation, ub_j, gs_j, body)$ de q_j tal que $p_j \in N_i$:

```
if not (in_error_i or found_i) then
  begin
    if (ub_j[j] = true) and (ub_i[i] = false) then
      in_error_i := true;
    if (ub_j[j] = true) and (ub_i[i] = undefined) then
      ub_i[i] := true;
    if (ub_j[j] = undefined) and (ub_i[i] = false) then
      for k := 1 to n do
        if gs_i[k] < gs_j[k] then
          begin
            gs_i[k] := gs_j[k];
            ub_i[k] := ub_j[k]
          end;
    if (ub_j[j] = undefined) and (ub_i[i] = undefined) then
      for k := 1 to n do
        if alt_gs_i[k] < gs_j[k] then
          alt_gs_i[k] := gs_j[k];
  end;
  Encaminhe (body) para p_i;
```

O algoritmo DETECT_UBP possui complexidade de mensagens de $O((c+nc)n\log T)$ *bits*, complexidade de tempo global de $O(n)$, complexidade de tempo local de $O(n)$ por mensagem recebida, e requer $O(n\log T)$ *bits* de memória por processo.

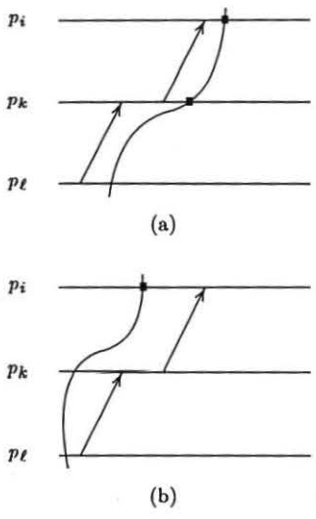

Figure 3: Estados globais mais adiantados e *breakpoints* incondicionais

5 Detecção de Predicados Conjuntivos

Nesta seção os algoritmos DETECT_STABLE_CP e DETECT_CP para detecção de predicados conjuntivos são discutidos. O primeiro é utilizado para detecção de predicados conjuntivos estáveis. Estes predicados são especificados para cada processo $p_i \in N$ como o predicado local lp_i favorecido com a propriedade de permanecer **true** uma vez que se torne **true**. No segundo algoritmo, tratamos de predicados conjuntivos genéricos, ou seja, não consideramos que os predicados locais são estáveis, e portanto todo predicado local pode se tornar falso e verdadeiro diversas vezes durante a computação. Como veremos, a estratégia para tratamento de propriedades instáveis torna o algoritmo mais difícil, embora tenhamos projetado este segundo algoritmo como uma extensão do primeiro. As principais contribuições deste algoritmo em relação aos algoritmos apresentados em trabalhos anteriores [4] [8] são os fatos deste ser totalmente distribuído e de não requisitar a gravação prévia de um histórico de execução.

5.1 Breakpoints sobre Predicados Conjuntivos Estáveis

Este algoritmo apresenta diversos aspectos semelhantes ao algoritmo DETECT_UBP. Na verdade, podemos considerar *breakpoints* incondicionais também como *breakpoints* sobre predicados conjuntivos estáveis, só que muito mais rígidos, pois no algoritmo DETECT_UBP o estado global detectado precisa combinar exatamente com os *breakpoints* incondicionais locais especificados pelos processos que participam do *breakpoint*. Por outro lado, o algoritmo que estamos tratando agora exige apenas que os predicados locais dos processos que participam sejam verdadeiros no estado global detectado, embora em alguns processos eles possam ter se tornado verdadeiros em tempos anteriores aos tempos locais dados pelo estado global. Portanto, o algoritmo apresentado nesta seção pode ser considerado uma simplificação do algoritmo DETECT_UBP, onde as condições de erro não precisam ser tratadas.

Como em muitos aspectos o algoritmo DETECT_STABLE_CP está relacionado com o algoritmo DETECT_UBP, este também pode ser visto como uma combinação dos princípios empregados nos algoritmos DETECT_DP e BROADCAST_WHEN_TRUE. Em relação ao último, a condição local lc_i para $p_i \in N$ é agora expressa pelo mesmo predicado local lp_i que já consideramos nos outros capítulos e a representação do estado global, gc_i, agora é dada pelo vetor cp_i. Para todo $p_k \in N$, $cp_i[k]$ é inicializado com **false** se p_k estiver participando no *breakpoint* e **true** caso contrário, da mesma forma que lp_k. Todas as outras variáveis empregadas pelo algoritmo DETECT_STABLE_CP têm o mesmo sentido que elas tinham quando usadas nos contextos anteriores.

A simplificação do algoritmo DETECT_UBP para produzir DETECT_STABLE_CP não vai além da eliminação da detecção de erro. O vetor alt_gs_i que fornece uma visão alternativa do estado global para q_i é ainda necessário para auxiliar na detecção do estado global de interesse mais adiantado. Da mesma forma que no caso de *breakpoints* incondicionais, uma cadeia causal de mensagens de *computação*, começando em q_ℓ tal que $cp_\ell[\ell]$ = **true** e atingindo q_i com $cp_i[i]$ = **false**, exige que q_i atualize gs_i de acordo com a informação anexada à mensagem de *computação* recebida. Por outro lado, se para o q_i atingido $cp_i[i]$ = **true**, existe uma chance de que os estados locais correspondentes aos envios de mensagens não contribuam para a detecção do estado global mais adiantado que satisfaz o predicado. Estes dois casos são ilustrados na Figura 4, respectivamente nas partes (a) e (b). Manter o alt_gs_i tem a função de permitir que este estado global mais adiantado seja salvo em gs_i, para ser usado no caso em que não haja cadeia causal do tipo que acabamos de descrever.

O vetor alt_gs_i é inicializado como gs_i e é anexado a mensagens de *computação* com seu i-ésimo componente modificado para lt_i. Uma mensagem de *computação* atingindo q_i afeta alt_gs_i e pode eventualmente afetar gs_i, o que acontece se $cp_i[i]$ = **false** na chegada da mensagem de *computação*, pela simples atualização de gs_i com alt_gs_i quando lp_i se torna **true**. Somente nesta situação, ou ao receber mensagens de *broadcast*, gs_i é atualizado. Neste último caso alt_gs_i também é atualizado. Assim $gs_i[k] \leq alt\_gs_i[k]$ para todo $p_k \in N$.

É mostrado a seguir o algoritmo DETECT_STABLE_CP que possui complexidade de mensagens de $O((c+nc)n\log T)$ bits, complexidade de tempo global de $O(n)$, complexidade de tempo local de $O(n)$ por mensagem recebida e requer $O(n\log T)$ bits de memória por processo.

Ações em q_i para o algoritmo DETECT_STABLE_CP:

(1) Ao detectar que lp_i se tornou **true**:

```
cp_i[i] := lp_i;
alt_gs_i[i] := lt_i;
for k := 1 to n do
  gs_i[k] := alt_gs_i[k];
if cp_i[1] ∧ ··· ∧ cp_i[n] then
  found_i := true
else
  Envie (broadcast, cp_i, gs_i, nil) para todo q_k tal que p_k ∈ N_i;
```

(2) Ao receber $(broadcast, cp_j, gs_j, \text{nil})$ de q_j tal que $p_j \in N_i$:

```
if not found_i then
  begin
    changed_i := false;
    for k := 1 to n do
      if gs_i[k] < gs_j[k] then
        begin
          gs_i[k] := gs_j[k];
```

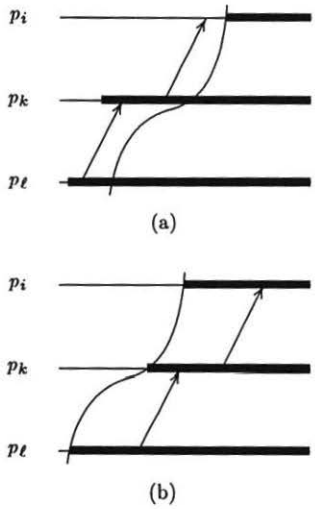

Figure 4: Estados globais mais adiantados e *breakpoints* sobre predicados conjuntivos estáveis

```
        cp_i[k] := cp_j[k];
        changed_i := true
      end;
  for k := 1 to n do
    if alt_gs_i[k] < gs_j[k] then
      alt_gs_i[k] := gs_j[k];
  if cp_i[i] and changed_i then
    if cp_i[1] ∧ ··· ∧ cp_i[n] then
      found_i := true
    else
      Envie (broadcast, cp_i, gs_i, nil) para todo q_k tal que p_k ∈ N_i
end;
```

(3) Ao receber $(body)$ de p_i destinado a $p_j \in N_i$:

Encaminhe $(computation, cp_i, alt\_gs_i(lt_i), body)$ para q_j;

(4) Ao receber $(computation, cp_j, gs_j, body)$ de q_j tal que $p_j \in N_i$:

```
if not found_i then
  for k := 1 to n do
    if alt_gs_i[k] < gs_j[k] then
      begin
        alt_gs_i[k] := gs_j[k];
        cp_i[k] := cp_j[k]
      end;
Encaminhe (body) para p_i;
```

5.2 Breakpoints sobre Predicados Conjuntivos Genéricos

Nesta seção são considerados os predicados conjuntivos, mas não mais consideramos que os predicados locais são estáveis, isto é, todo predicado local agora pode se tornar verdadeiro

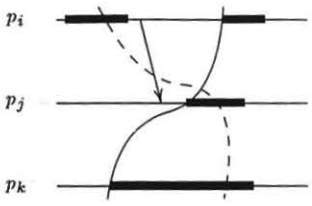

Figure 5: Estados globais e *breakpoints* sobre predicados conjuntivos genéricos

e falso ao longo da computação diversas vezes. Esta generalidade aparentemente agrava as dificuldades do problema, mas técnicas bastante similares às adotadas nas seções anteriores são suficientes como base para a solução empregada neste algoritmo. Nos algoritmos anteriores adotamos o vetor alt_gs_i como um meio de fornecer uma visão adicional em q_i do estado global a ser detectado, a fim de que gs_i pudesse guardar as características de um estado global mais adiantado a ser usado quando o vetor totalmente atualizado alt_gs_i não fosse necessário. Nesta seção este vetor também é empregado (com o mesmo objetivo), mas a maior complexidade do caso conjuntivo genérico requer que um vetor adicional, chamado alt_cp_i, seja também necessário para acompanhar o alt_gs_i. Ao contrário do caso estável, este vetor é necessário porque não há mais garantias de que os predicados locais permaneçam verdadeiros uma vez que tenham se tornado verdadeiros.

Assim, não somente precisamos de um meio de armazenar estados globais potencialmente mais adiantados para usá-los quando necessário, mas também devemos ter um meio de tratar a possibilidade de que os predicados locais possam se tornar verdadeiros e falsos diversas vezes antes de se tornarem verdadeiros em tempos locais com os quais eles possam participar em um estado global. Esta situação é descrita na Figura 5, na qual a partição indicada pela linha tracejada não é um estado global.

A solução para o algoritmo DETECT_CP é baseada em uma extensão do algoritmo DETECT_STABLE_CP para tratar a instabilidade de predicados locais. Esta extensão é baseada no fato de considerarmos que os canais em E sejam FIFO (First In, First Out), isto é, eles entregam mensagens na ordem em que elas foram enviadas. A questão central em obter esta extensão é assegurar que estados globais nos quais o predicado conjuntivo é satisfeito não sejam nunca perdidos. Está claro que isto não seria assegurado se, para $p_i \in N$, simplesmente acrescentássemos o novo vetor alt_cp_i ao algoritmo DETECT_STABLE_CP junto com uma nova ação para atribuir **false** a $alt\_cp_i[i]$ (e atualizar alt_gs_i de acordo) sempre que lp_i se tornasse **false**. Esta simples extensão não funcionaria mesmo na ausência de mensagens de *computação*, porque estados globais mais adiantados seriam perdidos. Se mensagens de *computação* fossem permitidas, a situação seria ainda pior, porque estados globais nos quais predicados conjuntivos são satisfeitos poderiam ser perdidos, fazendo desta forma a detecção impossível.

Neste algoritmo, impede-se que cp_i e alt_cp_i sejam atualizados quando q_i é atingido por um *broadcast* originado em q_k para $p_k \in N$ quando respectivamente, $cp_i[k] =$ **true** e $alt\_cp_i[k] =$ **true**. Isto inclui o caso em que $p_k = p_i$, isto é, $cp_i[i]$ e $alt\_cp_i[i]$ são somente atualizados se, respectivamente, $cp_i[i] =$ **false** e $alt\_cp_i[i] =$ **false**. Por esta razão, torna-se essencial para q_i conhecer a origem de um *broadcast* quando atingido pelas mensagens de *broadcast* correspondentes e então não se pode mais empregar, como foi feito, o algoritmo BROADCAST_WHEN_TRUE da Seção 3. A solução para o *broadcast* será então rotular as

mensagens com o tipo $broadcast_k$ para *broadcasts* originando em q_k.

A não atualização de cp_i e alt_cp_i quando uma mensagem de $broadcast_k$ é recebida (ou na origem do *broadcast*, se $p_k = p_i$) porque $cp_i[k]$ = **true** e $alt\_cp_i[k]$ = **true**, respectivamente, não provoca a perda de um estado global que satisfaça o predicado conjuntivo, como se poderia supor. Existe um aspecto relevante no algoritmo, onde a suposição de canais FIFO entra. Se o vetor gs_k acompanhando a mensagem de $broadcast_k$ fosse contribuir para o estado global detectado, então necessariamente uma cadeia causal de mensagens de computação partindo de q_k enquanto $alt\_cp_k[k]$ = **false** existiria destinado a algum q_j tal que $p_j \in N$, onde esta chegaria quando $alt\_cp_j[j]$ = **false**. Mas então uma das duas situações aconteceria envolvendo o gs_k que q_i ignorou. Se este gs_k chegasse a q_j antes que lp_j se tornasse **true**, então este seria levado em conta por q_j e participaria do *broadcast* que q_j geraria quando lp_j se tornasse **true**. Se, por outro lado, o gs_k chegasse em q_j depois que lp_j tivesse se tornado **true**, então seria perdido pelo *broadcast* iniciado por q_j na detecção de lp_j = **true**, mas seria levado em consideração por q_j e participaria no encaminhamento por q_j do *broadcast* iniciado por q_k. Entretanto, no último caso, só se poderia esperar que o gs_k fosse conduzido corretamente para todos os processos neste encaminhamento por q_j, se este fosse tratado como um novo *broadcast* iniciado em q_j. Então, além da necessidade já mencionada de anexar a identidade da origem do *broadcast* às mensagens de *broadcast*, os *broadcasts* que precisamos devem ser muito parecidos aos que empregamos anteriormente neste trabalho, no sentido de que uma mensagem de $broadcast_k$ chegando em q_i deve ser encaminhada se esta causar mudanças em $cp_i[k]$ ou $alt\_cp_i[k]$. As variáveis empregadas por q_i no algoritmo DETECT_CP são as mesmas empregadas no algoritmo DETECT_STABLE_CP, além do já mencionado vetor alt_cp_i. Eles são todos inicializados como no algoritmo anterior, e alt_cp_i é inicializado como cp_i, isto é, para $p_k \in N$ o k-ésimo componente é inicializado com **true** se p_k não participa do *breakpoint* ou **false** se p_k participa. Além disso, lp_i não se torna **false** se p_i não for um dos processos que participa do *breakpoint*.

Apresentamos a seguir o algoritmo para detecção de predicados conjuntivos genéricos.

Ações em q_i para o algoritmo DETECT_CP:

(1) Ao detectar que lp_i se tornou **true**:

```
if not found_i then
  begin
    alt_cp_i[i] := lp_i;
    alt_gs_i[i] := lt_i;
    Envie (broadcast_i, alt_cp_i, alt_gs_i, nil) para todo q_k tal que p_k ∈ N_i
  end;
```

(2) Ao detectar que lp_i se tornou **false**:

```
if not found_i then
  begin
    alt_cp_i[i] := lp_i;
    alt_gs_i[i] := lt_i
  end;
```

(3) Ao receber $(broadcast_\ell, cp_\ell, gs_\ell, \text{nil})$ de q_j tal que $p_j \in N_i$:

```
if not found_i then
  begin
    changed_i := false;
    if not cp_i[ℓ] then
      for k := 1 to n do
        if gs_i[k] < gs_ℓ[k] then
```

```
        begin
          gs_i[k] := gs_ℓ[k];
          cp_i[k] := cp_ℓ[k];
          changed_i := true
        end;
    if not alt_cp_i[ℓ] then
      for k := 1 to n do
        if alt_gs_i[k] < gs_ℓ[k] then
          begin
            alt_gs_i[k] := gs_ℓ[k];
            alt_cp_i[k] := cp_ℓ[k];
            changed_i := true
          end;
    if changed_i then
      if cp_i[1] ∧ ··· ∧ cp_i[n] then
        found_i := true
      else
        Envie (broadcast_ℓ, cp_ℓ, gs_ℓ, nil) para todo q_k tal que p_k ∈ N_i
  end;
```

(4) Ao receber $(body)$ de p_i destinado a p_j:

```
Encaminhe (computation, alt_cp_i, alt_gs_i(lt_i), body) para q_j;
```

(5) Ao receber $(computation, cp_j, gs_j, body)$ de q_j tal que $p_j \in N_i$:

```
if not found_i then
  for k := 1 to n do
    if alt_gs_i[k] < gs_j[k] then
      begin
        alt_gs_i[k] := gs_j[k];
        alt_cp_i[k] := cp_j[k]
      end;
Encaminhe (body) para p_i;
```

O algoritmo DETECT_CP possui complexidade de mensagens de $O((c + Pnc)n\log T)$ *bits*, onde P representa o número máximo de vezes que qualquer predicado local se torna verdadeiro, complexidade de tempo global de $O(n)$, complexidade de tempo local de $O(n)$ por mensagem recebida e requer $O(n\log T)$ *bits* de memória por processo.

6 Conclusões

Neste trabalho foi tratado o problema de projetar algoritmos distribuídos para a detecção de predicados globais em programas paralelos baseados em trocas de mensagens.

Em relação às contribuições deste trabalho, podemos citar como a principal o fato de os algoritmos projetados serem os primeiros totalmente distribuídos para os tipos de detecção de *breakpoints* que eles executam, enfatizando que eles detectam os estados globais mais adiantados com relação às propriedades envolvidas e que eles não requisitam que o processo tenha gerado em execução prévia um histórico de eventos. A detecção de forma distribuída foi possível, principalmente, devido à utilização de mensagens de *broadcast* que difundiam para todos os processos do sistema as visões das condições e tempos locais de um processo sempre que este tivesse seu predicado local satisfeito. A determinação do estado global mais adiantado foi obtida através da utilização de visões adicionais de tempos locais e condições que preservavam as informações necessárias para se obter o estado global mais adiantado que satisfizesse o predicado global.

Tendo em vista parar a computação no estado global detectado, sugerimos em [3] a utilização das técnicas de *checkpointing* e *rollback-recovery*. Claramente, necessitamos de memória adicional no emprego destas técnicas. Por isto apresentamos estratégias específicas para tratamento de *checkpointing* em cada um dos quatro casos. Neste mesmo contexto, sugerimos que estas estratégias, empregadas para economizar memória enquanto *checkpoints* são gravados, fossem também usadas quando desejamos avaliar expressões nos estados globais detectados pelos algoritmos e como resultado disso, precisamos gravar as variáveis envolvidas na expressão ao longo da execução do programa. Assim, podemos empregar as mesmas técnicas para economizar memória enquanto as variáveis são gravadas.

References

[1] Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Trans. on Computer Systems 3* (1985), 63–75.

[2] Cooper, R., and Marzullo, K. Consistent detection of global predicates. *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, 1991, pp. 167–174.

[3] Drummond, L. A., and Barbosa, V. C. Distributed Breakpoint Detection in Message-Passing Programs. *Publicações técnicas COPPE-UFRJ*, ES-306/94, submetido para publicação.

[4] Garg, V. K., and Waldecker, B. Detection of unstable predicates in distributed programs. *Proc. of the Twelfth Conference on Foundations of Software Technology & Theoretical Computer Science*, 1992, pp. 253–264.

[5] Goldberg, A. P., Gopal, A., Lowry, A., and Strom, R. Restoring consistent global states of distributed computations. *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, 1991, pp. 144–154.

[6] Haban, D., and Weigel, W. Global events and global breakpoints in distributed systems. *Proc. of the Twenty-First International Conference on System Sciences, Vol. 2*, 1988, pp. 166–175.

[7] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM 21* (1978), 558–565.

[8] Manabe, Y., and Imase, M. Global conditions in debugging distributed programs. *J. of Parallel and Distributed Computing 15* (1992), 62–69.

[9] McDowell, C. , and Helmbold, D. Debugging concurrent programs. *ACM Computing Surveys 21* (1989), 593–622.

[10] Miller, B., and Choi, J. Breakpoints and halting in distributed programs. *Proc. of the Eighth International Conference on Distributed Computing Systems*, 1988, pp. 316–323.

[11] Spezialetti, M. An approach to reducing delays in recognizing distributed event occurrences. In *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, 1991, pp. 155–166.

[12] Spezialetti, M., and Kearns, J. P. Simultaneous regions: a framework for the consistent monitoring of distributed systems. *Proc. of the Ninth International Conference on Distributed Computing Systems*, 1989, pp. 61–68.

[13] Tomlinson, A. I., and Garg, V. K. Detecting relational global predicates in distributed systems. *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993, pp. 21–31.