Detecção de Predicados Globais em Programas Paralelos Distribuídos

Lúcia Maria de Assumpção Drummond

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Valmir C. Barbosa, Ph.D.
(presidente)

Prof. Edil S. T. Fernandes, Ph.D.

Prof. Felipe M. G. França, Ph.D.

Prof. Orlando G. Loques Filho, Ph.D.

Prof. Osvaldo S. F. Carvalho, Dr.d'État

Prof. Virgílio A. F. Almeida, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 1994

Drummond, Lúcia Maria de Assumpção

Detecção de Predicados Globais em Programas Paralelos Distribuídos [Rio de Janeiro] 1994

XIII, 105 p., 29.7 cm, (COPPE/UFRJ, D. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1994)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Predicados Globais 2 – Estados Globais 3 – Sistemas Distribuídos

I. COPPE/UFRJ II. Título(Série).

A Irapuan e meus pais

Agradecimentos

Ao meu orientador Valmir, por quem tenho grande admiração, por estes quase sete anos de ensinamentos, durante o mestrado e o doutorado, por seu exemplo de dedicação à vida acadêmica e também pela convivência agradável.

À Anna pela amizade demonstrada nos muitos momentos de estudo e trabalho compartilhados.

Ao Satoru por sua cordial disponibilidade e oportunas observações durante a fase de redação da tese.

À Clícia e à Priscila pelo apoio e incentivo durante a fase final do doutorado, pelas sugestões no texto da tese, e também pelas horas alegres e de descontração.

Ao Ricardo Bianchini e à Cristina Boeres, que estão em doutoramento no exterior, pelas cópias de alguns artigos referenciados nesta tese.

Ao Departamento de Computação da Universidade Federal Fluminense pelo apoio durante todo o curso de doutorado.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

Detecção de Predicados Globais em Programas Paralelos Distribuídos

Lúcia Maria de Assumpção Drummond

Dezembro de 1994

Orientador: Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

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.

Nesta tese 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.

Objetivando parar a computação no estado global exato que os algoritmos detectam, sugerimos o uso das técnicas de *checkpointing* e *rollback-recovery* e indicamos para cada um dos quatro casos como realizar a gravação de *checkpoints* a fim de que a memória adicional requerida por cada processo não seja maior do que a necessidade real.

Uma outra questão relacionada à detecção de predicados consiste na avaliação de expressões (composta de variáveis de diferentes processos) no estado detectado. A estratégia sugerida para gravação das variáveis que compõem a expressão é semelhante à empregada na gravação de *checkpoints.*

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Doctor of Science (D. Sc.)

Global Predicate Detection in Distributed Parallel Programs

Lúcia Maria de Assumpção Drummond

December, 1994

Thesis Supervisor: Valmir C. Barbosa

Department: Programa de Engenharia de Sistemas e Computação

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 thesis 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 ac-

tually 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.

In order to actually halt the computation at the exact global state the algorithms detect, we suggest the use of checkpointing and rollback recovery techniques. Also, we indicate for each of the four cases how to approach the recording of checkpoints so that the additional storage requirement per process is not demanding beyond the actual needs.

Another issue related to global predicate detection consists of expression evaluation in the global state detected. Having in mind that an expression is composed by variables from several processes, we suggest the same strategy used for recording checkpoints as a mechanism for recording variables.

Índice

Lista de Figuras

Capítulo 1

Introdução

1.1 Apresentação

Um programa paralelo distribuído consiste de um conjunto de processos que trabalham juntos para realizar uma tarefa e que se comunicam apenas por troca de mensagens. O *software* distribuído, como qualquer outro, está sujeito a erros que poderiam ser localizados e analisados através de um depurador, conforme ocorre em programas seqüenciais. Entretanto, há várias razões para se considerar a depuração de programas paralelos distribuídos muito mais difícil do que a de programas seqüenciais. Dentre elas podemos citar [18]:

- A determinação do estado do sistema distribuído em um dado instante é difícil devido à inexistência de um relógio comum aos processadores.
- Sistemas distribuídos assíncronos não são determinísticos. Isto significa que duas execuções do mesmo sistema podem produzir ordenações de eventos diferentes, o que torna difícil a reprodução de erros e testes de

situações possíveis, mas improváveis.

- A monitoração de um sistema distribuído altera o seu comportamento, o que é chamado de *probe-effect*. O comportamento de um programa seqüencial não é afetado pelo intervalo de tempo entre a execução de duas instruções sucessivas. Assim, um depurador pode interromper um processo seqüencial em um ponto sem afetar a execução subseqüente do processo. Em um sistema distribuído, parar ou tornar mais lento um processo pode alterar o comportamento do processo inteiro.

- Um sistema distribuído não é facilmente controlável. Assim, técnicas de depuração, tais como *tracing* e *breakpoints*, baseadas em um contador de programa e um estado do processo, precisam ser estendidas para serem aplicáveis a sistemas distribuídos.

- A interação entre o sistema e o usuário das ferramentas de monitoração, pode ser mais complexa. Por exemplo, quando um terminal é conectado a cada processador, o usuário pode precisar mudar fisicamente de terminal para terminal a fim de inicializar processos, estabelecer *breakpoints* e examinar *traces*. Portanto, é necessário fornecer ferramentas que possam ser chamadas e controladas de um único lugar.

Neste contexto, o problema de depuração de programas distribuídos pode ser tratado, principalmente, de duas perspectivas que, embora diferentes, são complementares, conforme podemos observar nas principais contribuições que têm aparecido ultimamente na literatura da área [4, 10–14, 16, 22, 23, 27–29].

A primeira possibilita a repetição determinística de programas paralelos que não são determinísticos, objetivando a reprodução exata do comportamento observado na execução anterior. Esta perspectiva permite a utilização da técnica de depuração cíclica, empregada em programas seqüenciais, também em programas paralelos. Esta técnica consiste em parar o programa repetidamente durante a execução, examinando o seu estado e então continuando ou reexecutando-o a fim de parar o programa em um outro ponto. A sua eficiência se deve ao fato de que programas seqüenciais são normalmente determinísticos, isto é, para uma entrada fixa, cada execução de um programa seguirá sempre o mesmo caminho de execução e produzirá os mesmos resultados. Em programas paralelos, esta questão é contornada através da gravação de um histórico de execução (também chamado de arquivo de *trace*), sobre o qual são baseadas as execuções subseqüentes, permitindo assim a reprodução da execução original.

A segunda perspectiva se refere ao uso de *breakpoints*, que são aplicados para parar a execução de um programa em localizações específicas a fim de examinar seu estado. Em depuradores seqüenciais, os *breakpoints* geralmente correspondem a estados do processo onde um predicado é satisfeito, em função da ocorrência de algum evento. Em depuradores de programas paralelos distribuídos, o conjunto de predicados usado para expressar *breakpoints* é maior, pois, neste caso, os predicados podem ser representados por condições que dependem de eventos de múltiplos processos, sendo denominados, então, predicados globais. Enquanto a determinação do estado de um programa seqüencial que satisfaz um predicado é muito simples, em um programa paralelo distribuído ela não é trivial, pois os processos não podem determinar seus estados no mesmo tempo, devido à ausência de um relógio

global. Portanto, a determinação de um estado global (que é composto de estados locais, um para cada processo, e do conjunto de mensagens em trânsito para cada canal) que satisfaça um predicado em programas paralelos distribuídos é uma questão bastante complexa.

Dentro deste largo espectro de problemas associados com a depuração de sistemas distribuídos, nesta tese nos concentramos no projeto e análise de algoritmos distribuídos para detecção de ocorrência de *breakpoints*. Os *breakpoints* que consideramos são distribuídos e são expressos por uma coleção de condições inter-relacionadas espalhadas por todo o sistema. Neste trabalho, eles podem ser de três tipos. Os *breakpoints* incondicionais são pontos pré-estabelecidos especificados distribuidamente entre os componentes do programa. Os *breakpoints* condicionais que consideramos são expressos por predicados conjuntivos e disjuntivos. Os predicados conjuntivos especificam uma condição global através de uma conjunção lógica de um conjunto de condições locais (predicados locais) espalhadas pelo sistema, enquanto que em predicados disjuntivos a condição global é dada pela disjunção lógica das 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 para detecção de ocorrência de tais *breakpoints* nesta tese são capazes de informar os estados globais mais adiantados que satisfazem os *breakpoints*.

Se desejássemos parar a computação no estado global detectado pelos algoritmos citados acima, seria necessária a propagação de uma ordem para que os processos parassem suas execuções quando esta detecção ocorresse.

Além disso, algum mecanismo de retorno teria que ser utilizado, já que estas mensagens atingiriam os processos em tempos mais adiantados do que os tempos locais com os quais eles participaram da detecção. Por esse motivo, é necessário empregar técnicas de *checkpointing* e *rollback-recovery* [20]. Assim, ainda discutimos neste trabalho estratégias adicionais para se economizar memória, enquanto os *checkpoints* são gravados. Estas estratégias também podem ser úteis no tratamento de um outro problema bastante comum em depuração. Este problema se refere à avaliação de expressões em *breakpoints*, ou melhor, todos os predicados globais, para os quais projetamos algoritmos, podem ser utilizados para definir estados globais onde se deseje avaliar uma determinada expressão, a qual pode ser constituída de variáveis de todos os processos que participam da computação. Para isto, as variáveis devem ser salvas diversas vezes no decorrer da computação quando mudam de valor. As estratégias propostas para se economizar memória nas gravações de *checkpoints* e variáveis são semelhantes.

Poucos são os autores que tratam das mesmas questões que são apresentadas nesta tese, conforme veremos em um capítulo mais adiante, e estes propõem algoritmos que apresentam pelo menos um dos seguintes problemas, que nos nossos algoritmos foram contornados: são centralizados, não detectam o estado global mais adiantado em que o *breakpoint* é satisfeito, não são eficientes em relação à complexidade ou ainda precisam utilizar um histórico de execução para realizar a detecção, o que aumenta de forma considerável os requerimentos de memória do algoritmo.

A principal contribuição desta tese consiste no projeto e análise de quatro algoritmos distribuídos para a detecção dos estados globais mais adiantados

que satisfaçam *breakpoints* (incondicionais ou condicionais).

Em relação aos *breakpoints* incondicionais, projetamos o algoritmo DETECT_UBP que detecta, quando possível, o estado global mais adiantado composto por pontos pré-estabelecidos de cada processo que participa do *breakpoint* incondicional. Não encontramos nos trabalhos relacionados projetos de algoritmos corretos para este tipo de detecção.

Quando consideramos *breakpoints* condicionais, examinamos os expressos por predicados disjuntivos e conjuntivos. Nesta tese, projetamos o algoritmo DETECT_DP, usado para detectar o estado global mais adiantado que satisfaz um *breakpoint* expresso como predicado disjuntivo. Os autores dos trabalhos relacionados que tratam de detecção de predicados disjuntivos propõem algoritmos que não são capazes de detectar o estado global mais adiantado que satisfaça tal predicado.

Em relação aos predicados conjuntivos, apresentamos dois algoritmos. O primeiro DETECT_STABLE_CP é utilizado para detecção de predicados conjuntivos estáveis. No segundo algoritmo, DETECT_CP, tratamos de predicados conjuntivos genéricos, ou seja, os predicados locais não são estáveis e podem se tornar falsos e verdadeiros diversas vezes durante a computação. A maior parte dos algoritmos existentes nos trabalhos analisados trata de predicados estáveis e não consegue detectar os estados globais mais adiantados para este tipo de *breakpoint*. Quando examinamos os algoritmos existentes para detecção de predicados conjuntivos genéricos, observamos que eles ou são centralizados ou requisitam que um histórico de execução tenha sido gerado previamente.

Outra contribuição desta tese se refere às estratégias para economia de memória acrescentadas às técnicas de *checkpoint* e *rollback-recovery* utilizadas para parar o programa no estado global detectado. Estas estratégias também podem ser empregadas para economizar memória na gravação de variáveis visando análise de expressões em *breakpoints*.

1.2 Organização da Tese

O restante deste trabalho está organizado da seguinte forma:

Antes dos algoritmos serem apresentados, introduzimos três capítulos. Algumas informações a respeito das técnicas mais comuns de depuração de programas paralelos são apresentadas no Capítulo 2, onde, inclusive, nos aprofundamos um pouco mais na perspectiva citada inicialmente neste capítulo, que permite a repetição determinística de programas paralelos (que não são determinísticos) a fim de reproduzir, em cada execução, o mesmo comportamento. No Capítulo 3 é definido o modelo básico do sistema distribuído e são feitas algumas considerações sobre a notação utilizada pelos algoritmos e os principais conceitos que envolvem o assunto.

No Capítulo 4 são apresentados os trabalhos anteriores, com ênfase naqueles que tratam de questões intimamente relacionadas com as que estudamos nesta tese.

Nos capítulos seguintes são apresentados os algoritmos que constituem a contribuição desta tese e são estabelecidas, através de teoremas, a corretude e a complexidade destes.

No Capítulo 5, apresentamos o algoritmo DETECT_DP, usado para detectar *breakpoints* expressos como predicados disjuntivos. Este algoritmo emprega somente um tipo de mensagens, as da própria aplicação, chamadas neste trabalho de mensagens de *computação*. Neste capítulo, fazemos também algumas considerações que serão usadas nos capítulos seguintes. Apresentamos, o algoritmo BROADCAST_WHEN_TRUE que detecta *breakpoints* expressos como predicados conjuntivos em aplicações que não trocam mensagens. Assim, somente mensagens especiais chamadas de mensagens de *broadcast* são utilizadas pelo algoritmo. O mesmo tem como objetivo principal simplificar a compreensão dos demais algoritmos que serão apresentados nos capítulos seguintes, já que estes outros algoritmos estão entre os dois extremos dados por DETECT_DP e BROADCAST_WHEN_TRUE.

No Capítulo 6, o algoritmo DETECT_UBP para a detecção de *breakpoints* incondicionais é apresentado. Este algoritmo é o único que precisa informar um erro, caso o *breakpoint* incondicional, que é especificado como um conjunto de tempos locais, um para cada processo participante, não constitua um estado global. Em todos os outros algoritmos, apresentados nos demais capítulos, caso não exista nenhum estado global no qual o predicado seja satisfeito, o programa simplesmente não deve parar até que chegue ao fim.

Os algoritmos para detecção de predicados conjuntivos DETECT_STABLE_CP e DETECT_CP são apresentados no Capítulo 7. O primeiro, utilizado para detecção de predicados conjuntivos estáveis, em muitos aspectos pode ser considerado uma simplificação do algoritmo DETECT_UBP, apresentando inclusive as mesmas complexidades de tempo e mensagens, assim como a necessidade de memória . No segundo algoritmo,

tratamos de predicados conjuntivos genéricos, ou seja, os predicados locais não são estáveis o que, como será visto, aumenta bastante a dificuldade do problema.

As técnicas de *checkpointing* e *rollback-recovery* empregadas para parar o programa no estado global detectado são descritas no Capítulo 8, onde são também discutidas estratégias adicionais para economizar memória enquanto os *checkpoints* são gravados. Estas estratégias, como já exposto, também podem ser úteis na avaliação de expressões em *breakpoints* a fim de economizar memória nas gravações das variáveis. Em função disso, este tópico também é tratado no Capítulo 8.

Finalmente, o Capítulo 9 conclui a tese apresentando um resumo da pesquisa realizada e sugestões para futuros trabalhos.

Capítulo 2

Técnicas de Depuração

O objetivo deste capítulo é descrever brevemente as principais técnicas de depuração de programas paralelos.

Conforme relatado no Capítulo 1, existem basicamente duas perspectivas no tratamento de depuração de programas paralelos. A primeira permite a repetição determinística de um programa paralelo através da gravação de um histórico de eventos. Na segunda podemos estabelecer *breakpoints* que nos permitam fazer uma análise do programa em localizações específicas.

A primeira perspectiva faz parte de um conjunto de técnicas de depuração que serão abordadas neste capítulo. Na seção 2.1, serão tratadas as técnicas que utilizam históricos de eventos, que são as mais citadas em trabalhos na área, enquanto na seção 2.2 faremos algumas observações, mais breves que as da seção anterior, sobre outras técnicas existentes, como técnicas de depuração convencional aplicadas a programas paralelos e técnicas de análise estática, que permitem que alguns erros sejam detectados sem a execução do programa.

A segunda perspectiva será tratada no decorrer deste trabalho nos capítulos seguintes.

2.1 Histórico de Eventos

As ferramentas de depuração de programas paralelos deveriam ser úteis na descoberta de erros resultantes da interação de vários processos. Em muitos casos, estes erros ocorrem somente quando os processos trocam dados e sincronizam suas tarefas em uma dada ordem. Como os resultados observados são difíceis de serem reproduzidos, a gravação de um histórico de eventos (arquivo de *trace*) pode ter um papel bastante útil. O *trace* possui uma entrada para cada evento importante que ocorre na vida do processo. A definição de evento varia de sistema para sistema. Um evento pode ser um acesso à memória, um envio ou recebimento de mensagem, um acesso a um objeto ou pode mesmo ser definido pelo programador.

A quantidade de informação a ser gravada para cada evento depende de como o histórico de eventos vai ser usado [28]. Apesar de inicialmente considerarmos o histórico de eventos importante apenas para controlar a reexecução do programa (técnica chamada de *replay*), este ainda pode ser utilizado simplesmente para exibição de eventos ou em um caso mais complexo para simular o ambiente de qualquer processo, permitindo o uso de um depurador seqüencial em um processo sem a reexecução de todo o programa (*simulação*).

Nos sistemas que visam meramente a exibição dos eventos, a quantidade de informação a ser gravada para cada evento pode ser mínima.

Em sistemas em que o depurador usa o histórico de eventos para controlar a reexecução do programa, permitindo o uso de técnicas de depuração convencionais, tais como *breakpoints* e execução passo a passo sem a modificação do comportamento do programa, é necessário que informações suficientes sejam gravadas possibilitando a determinação do evento em que cada processo participará.

Quando o histórico de eventos é utilizado para simular o ambiente de qualquer processo, todos os eventos visíveis ao processo são gravados, requerendo mais informações gravadas do que os casos anteriores.

Existem dois tipos de ferramentas que seriam importantes para depuração baseada em um histórico de eventos: a primeira seria para produzir o *trace*, a segunda para utilizá-lo.

Ao considerarmos a primeira ferramenta, observamos que, dependendo do tipo de evento, o arquivo de *trace* contém alguns parâmetros padrões e alguns parâmetros específicos dos eventos, onde os parâmetros padrões podem ser [10]:

- Tipo da entrada: este parâmetro define o tipo do evento (por exemplo: troca de mensagem).
- Identificação do processo: em muitos casos, é conveniente armazenar todos os *traces* de um dado nó em um único arquivo. Como as entradas para os vários processos podem ser intercaladas, é necessário identificar o processo que executa o evento.
- Timestamp: o *timestamp* é um número que identifica unicamente o

tempo em que o evento ocorreu. Os *timestamps* fornecem uma ordenação dos eventos do sistema, e podem ser gerados por relógios lógicos.

- Identificação da transação (ou *job*): uma transação é uma coleção de ações relacionadas que são tratadas pelo sistema como uma unidade. Em uma base de dados distribuída, uma transação pode ser um pedido para aumentar um salário de todos os empregados em dez por cento. Um *job* para imprimir um arquivo em uma localização remota também é uma transação.

Em relação aos parâmetros específicos, podemos exemplificar com o evento "envio de mensagem", onde os parâmetros específicos poderiam ser as identificações dos processos origem e destino.

Além da quantidade de informação a ser gravada, há também a questão de como a gravação deve ser feita e o impacto resultante no desempenho e *probe effect*. Para que a gravação seja feita, podemos inserir comandos apropriados no programa fonte original, ou as rotinas do sistema podem ser modificadas ou ainda pode-se fazer o *link* do programa usando-se versões especiais de monitoração. Todos estes casos são suscetíveis ao *probe effect*.

Quando se considera a segunda ferramenta, observamos que pode-se utilizar o histórico de eventos de várias formas. Após as gerações dos *traces* o programador pode exibi-los e identificar a interação do processo que produziu o erro. Neste caso, algumas vezes o programador é capaz de identificar precisamente a causa do erro apenas através das informações nos *traces*. Em outros casos, porém, terá que tentar recriar as condições sob as quais o erro original foi observado, através da reexecução controlada ou através da simu-

lação. A seguir discutiremos cada uma destas alternativas.

No caso de utilização de histórico de eventos visando apenas a exibição, é reconhecida a necessidade de se facilitar a interpretação do histórico de eventos que se apresenta sob diversas formas, utilizando-se inclusive técnicas gráficas para este fim. Uma outra proposta neste sentido consiste no armazenamento do histórico de eventos em um banco de dados.

Dentre as desvantagens de se utilizar o histórico de eventos apenas para exame podemos citar: não existe mecanismo para se obter informação adicional sobre um erro que foi observado, a quantidade de informação tende a ser volumosa perdendo-se muitas vezes resultados importantes, e, como a técnica não é cíclica, o usuário deve coletar informação suficiente durante a execução para diagnosticar qualquer erro que poderia surgir.

Uma outra alternativa é baseda na reprodução da execução de programas, permitindo que técnicas de depuração cíclica sejam aplicadas (por exemplo [22] [31] [32]).

A forma mais simples de se usar depuração cíclica é através da colocação de comandos de saída em um programa com erro de forma a se obter detalhes adicionais sobre a execução do programa. Execuções sucessivas podem ser usadas para se obter maiores detalhes sobre as partes do programa sob suspeita.

Quando se trata de depuração de programas seqüenciais, pode-se normalmente garantir a reprodução da execução do programa fornecendo-se a mesma entrada cada vez que o programa é executado. Execuções sucessivas com as mesmas entradas produzem o mesmo comportamento porque progra-

mas seqüenciais são determinísticos. Não estamos considerando, neste caso, programas probabilísticos. O mesmo é verdadeiro para processos individuais em um programa paralelo. Se a cada processo é fornecida a mesma entrada (correspondentes aos conteúdos das mensagens recebidas) na mesma ordem durante execuções sucessivas, teremos sempre o mesmo comportamento. Em particular, cada processo produzirá os mesmos valores de saída na mesma ordem. Cada um destes valores de saída pode então servir como entrada para um outro processo. Por isso, para se depurar um programa paralelo, não é necessário gravar todos os valores de entrada no histórico de eventos, pois todo valor de entrada correspondente a algum valor de saída pode ser recalculado durante a reexecução. Assegurando-se que cada processo obtenha os mesmos valores de entrada em todo passo da execução, todos os processos exibirão o mesmo comportamento na sua reexecução. Portanto, utilizando-se a técnica de reexecução controlada, comandos de saída que poderiam mudar a temporização relativa de operações no programa, não produzirão uma seqüência de execução diferente.

Resumindo, esta facilidade pode ser implementada da seguinte forma: Após a gravação dos eventos em um arquivo, o programa é reexecutado com um *controlador* que usa o arquivo de histórico (*trace*) para guiar a reexecução do sistema. Este controlador obtém os eventos do arquivo e os compara com as ações no sistema.

Em [18], quando uma *primitiva* é requisitada por um processo, um *send* por exemplo, onde o processo transmissor fica bloqueado até receber uma mensagem de resposta, uma mensagem é enviada para o processo controlador enquanto o processo que requisitou a primitiva fica bloqueado até re-

ceber uma resposta. Usando-se o controlador, podemos atrasar a resposta ao processo de aplicação. Assim, em um dado instante, existe um conjunto de eventos pendentes que estão sendo atrasados pelo controlador. A todo momento o controlador sabe qual é o próximo evento a ser executado para assegurar que esta execução combine com a execução original. Deste modo, o controlador espera até que este evento ocorra ou que um processo faça alguma coisa diferente do que o histórico indica, ou o usuário entre com um comando que mude a execução do sistema ou ainda que o usuário desligue o modo de reexecução controlada. Nos demais casos, o sistema pode continuar a execução.

Esta técnica de reexecução controlada permite o uso de *breakpoints*. A inserção de um *breakpoint* em um processo de um programa paralelo pode mudar a ordem relativa de eventos durante a execução, produzindo uma seqüência de execução diferente a cada vez. Entretanto, usando-se o histórico de eventos, podemos reproduzir a execução, mesmo na presença de *breakpoints*, pois as operações são ordenadas conforme o histórico de eventos.

A técnica de reexecução controlada requisita que a entrada de cada processo seja recalculada durante a reexecução, ao invés de ser recuperada do histórico de eventos. Esta técnica apresenta vantagens e desvantagens, pois se por um lado na fase de monitoração gasta-se menos tempo e espaço, por outro lado, requer que todos os processos sejam reexecutados na segunda fase. A reexecução global é uma desvantagem se as necessidades computacionais para reexecutar um programa são muito grandes, particularmente quando é desnecessário recriar o conjunto inteiro de processos originais para se isolar um erro. Usando-se um histórico de eventos mais detalhado, pode-se reexe-

cutar o subconjunto de processos nos quais se está interessado e simular o resto. Na verdade, estamos deslocando o custo da fase de geração do histórico de eventos para a fase de reexecução. Quando desejamos reexecutar um subconjunto de processos de uma computação freqüentemente, como acontece no caso de utilização de depuração cíclica para isolar um erro, coletamos informação adicional no histórico de eventos para eliminar a necessidade de reexecução do programa inteiro. As informações adicionais a serem gravadas seriam as entradas para o subconjunto de processos de interesse. Nas execuções subseqüentes, somente o subconjunto de processos designados seria reexecutado e sua interface com o ambiente externo seria simulada usando o histórico de eventos.

2.2 Outras técnicas

A técnica de depuração que utiliza histórico de eventos descrita acima é a mais referenciada em termos de depuração de programas paralelos [4]. Entretanto, existem outras, que serão descritas a seguir.

A forma mais simples de implementar um depurador paralelo é construí-lo como uma coleção de depuradores seqüenciais, um por processo paralelo, onde o depurador paralelo é apenas uma extensão de um depurador para programas seqüenciais.

A principal diferença em relação ao depurador seqüencial está na forma de exibir a saída dos diversos depuradores e nos seus controles. Em relação à exibição da saída, de uma forma mais primitiva, o que necessitamos é fornecer múltiplos terminais reais ou virtuais ao usuário de um ambiente paralelo

para que se possa executar as múltiplas cópias do depurador. As atuais estações de trabalho de múltiplas janelas tornam isto mais prático [35] [41]. Assim, com um gerenciador de janelas poderíamos direcionar um comando do depurador para uma tarefa específica e também diferenciar as saídas das diversas tarefas no terminal. Para direcionar o comando do depurador para um conjunto arbitrário de tarefas repetiríamos o comando desejado em cada uma das janelas.

Estes depuradores paralelos tradicionais normalmente suportam os mesmos tipos de *breakpoints* que aqueles achados em depuradores seqüenciais. Estes *breakpoints* são locais e incluem parada em um comando, parada na ocorrência de uma exceção ou algum evento detectável pelo usuário, parada quando uma variável específica é acessada, e parada quando uma expressão condicional é satisfeita [36]. Neste caso existem duas possibilidades quando um *breakpoint* é encontrado. Ou todos os processos no programa paralelo são parados ou somente o processo que encontrou o *breakpoint* é parado. A primeira possibilidade pode ser difícil de atingir dentro de um intervalo de tempo suficientemente pequeno, e a última pode causar um sério impacto em sistemas que contenham *timeouts*. A estes depuradores paralelos poderíamos acrescentar a facilidade de estabelecer *breakpoints* descritos por eventos espalhados pelo sistema.

Outra técnica que pode ser utilizada em depuração é a de análise estática [27] [42], onde o processo não precisa ser executado para detecção de erros, o que é feito através da análise do programa. Esta técnica é utilizada para detectar basicamente dois tipos de erros em programas paralelos, ou sejam, erros de sincronização e erros de uso de dados. Erros de sincronização in-

cluem os problemas de *deadlock* e *starvation*. Erros de usos de dados incluem os erros comuns de programas seqüenciais tal como leitura de uma variável não inicializada e erros comuns a programas paralelos, como acesso simultâneo a uma variável compartilhada (sistemas de memória compartilhada). O resultado da aplicação da análise estática é um relatório de erros.

Existem algumas propostas para combinar análise estática com depuração dinâmica, como [1] [15] [30].

Por exemplo, se pudéssemos mostrar que partes de um programa estão livres de erros usando análise estática, então estas partes estariam livres de monitoração. Isto poderia reduzir o *overhead* associado com a monitoração. Por outro lado, se no programa existem expressões executadas condicionalmente e se já se soubessem os resultados das condições, o número de estados a serem examinados pelo analisador estático seria reduzido.

Outra idéia é eliminar os erros durante o desenvolvimento do programa [2][3]. Por exemplo, os compiladores de vetorização automática geram programas paralelos corretos. Outros determinam quando um *loop* pode ser paralelizado. Ou seja, é possível depurar uma versão seqüencial de um programa usando ferramentas de depuração convencionais e então transformar esta em uma versão paralela equivalente.

Em todas estas técnicas é desejável que sejam apresentadas informações ao usuário para que este possa detectar os erros. Neste contexto, um importante assunto de pesquisa é "visualização", que auxilia o programador a visualizar o comportamento de uma aplicação ou sistema através da apresentação de seu estado e progresso da sua execução graficamente.

Em [28] são apresentadas três técnicas de visualização objetivando a depuração, que podem também ser utilizadas para análise de *performance*.

O tipo mais comum de exibição de informações é baseado na apresentação seqüencial de um simples texto de informações (por exemplo [25]).

Outra técnica utilizada para depuradores baseados em histórico de eventos é o diagrama tempo-processo, que consiste em uma representação bidimensional do estado de um sistema paralelo em relação ao tempo, sendo possível observar a cada instante em quais eventos os processos participam (por exemplo [14][23] [40]).

Existe ainda a possibilidade de se utilizar animação com a exibição de "fotografias" dos estados do programa (como [16] [37]).

Capítulo 3

Modelo e Notação

3.1 Estados Globais e Breakpoints

Neste trabalho, o programa a ser depurado é representado pelo grafo não direcionado $G = (N, E)$, com $n = |N|$ e $e = |E|$. 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. Vejamos a definição da relação binária *ANTES* sobre V [21]. Dizemos que v_1 *ANTES* v_2, onde $v_1, v_2 \in V$, se e somente se uma das condições a seguir é satisfeita:

- os eventos v_1 e v_2 ocorrem no mesmo processo, p_i por exemplo, nos tempos locais t_i^1 e t_i^2, respectivamente, sendo que $t_i^1 < t_i^2$ e nenhum outro evento ocorre naquele mesmo processo num instante t_i' tal que $t_i^1 < t_i' < t_i^2$.
- v_1 envolve o envio de uma mensagem m recebida através de v_2, sendo que v_1 e v_2 ocorrem em processos vizinhos.

Seja A o fecho transitivo de *ANTES*, isto é, se $v, v' \in V$, então $(v, v') \in A$ se e somente se, para algum $r \geq 0$ existem r eventos $v_1, \cdots, v_r$ tais que

$$v \ \textit{ANTES}\ v_1,$$
$$v_1 \ \textit{ANTES}\ v_2,$$

etc, até

$$v_r \; ANTES \; v'.$$

A relação A é irreflexiva e transitiva, formando portanto uma ordem parcial sobre os eventos de V. A relação A, também chamada de "aconteceu antes", é usada na definição de "estados globais consistentes" ou mais sucintamente "estados globais". Um estado global é uma partição (V_1, V_2) de V tal que $v \in V_1$ somente se todo $v' \in V$ tal que $v'Av$ é também um membro de V_1 [6]. Claramente, em um estado global não existe nenhum par $(v, v') \in A$ tal que $v \in V_2$ e $v' \in V_1$.

O problema de detectar um *breakpoint* e parar o programa corretamente deveria então ser proposto como o problema de detectar um estado global (e então parar o programa) no qual a condição que especifica o *breakpoint* é satisfeita.

Algoritmos para resolver o problema de detecção de *breakpoint* não devem nunca perder um estado global que satisfaça a condição estabelecida. Em termos de estados globais, um *breakpoint* incondicional é especificado como um conjunto de tempos locais, um para cada processo participante (consideraremos que existe pelo menos um), para um subconjunto de N. Se esta especificação não constituir um estado global, o algoritmo deve informar um erro. No decorrer do trabalho iremos nos referir ao tempo local que especifica o *breakpoint* incondicional, em cada um dos processos que participa da computação, por *breakpoint* incondicional local, denotado por lub_i para $p_i \in N$. *Breakpoints* condicionais, por sua vez, serão especificados como o conjunto de predicados locais, um para cada processo participante (também consideraremos que existe pelo menos um). Se não existir nenhum estado

global no qual o predicado é satisfeito, então o programa simplesmente não deverá parar até que chegue ao fim. O predicado local associado com p_i é referenciado como a variável lógica lp_i.

Freqüentemente, em relação a *breakpoints* condicionais, 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. Formalizando, o estado global (V_1', V_2') é mais adiantado do que um estado global (V_1, V_2) se e somente se $V_1' \subset V_1$. Esta definição permite que mais de um estado global mais adiantado exista em relação à mesma propriedade, por exemplo (V_1, V_2) e (V_1', V_2') tal que não ocorra $V_1 \subset V_1'$ nem $V_1' \subset V_1$. Se o predicado sob consideração é um predicado estável, então, os outros estados globais nos quais ele é satisfeito, além de (V_1, V_2), incluem todos os estados globais (V_1', V_2') tal que $V_1 \subset V_1'$.

No caso de *breakpoints* incondicionais, a detecção de um estado global mais adiantado só tem sentido se existir no mínimo um processo que não participe do *breakpoint*. Os processos para os quais não são especificados *breakpoints* incondicionais locais devem participar no estado global detectado com os estados locais mais adiantados possíveis.

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 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 $p_k \in N$.

3.2 Ilustrações

Pode-se descrever as ocorrências de eventos sobre o tempo através de um diagrama de tempo. Assim, os processos são representados por eixos de tempos locais, nos quais os tempos locais crescem da esquerda para a direita e as setas representam mensagens. Nesta representação, que será utilizada no decorrer do trabalho, um estado global é um corte (linha) dividindo o

diagrama de tempo em duas partes. Informalmente, um corte (estado global) no diagrama de tempo é consistente se nenhuma seta parte do lado direito do corte e chega do lado esquerdo deste. Esta noção de consistência adequa-se à observação de que uma mensagem não pode ser recebida antes de ser enviada. Por exemplo, os cortes nas partes (a) e (b), da Figura 3.1, são cortes consistente e inconsistente, respectivamente.

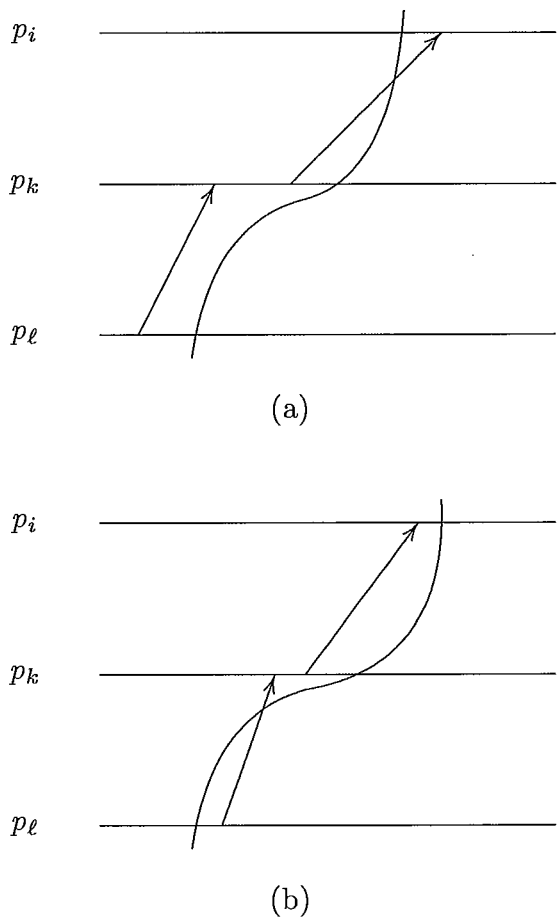

Figura 3.1: Estados globais consistente e inconsistente

Usando esta mesma representação, na Figura 3.2 o estado global representado pela linha tracejada é mais adiantado do que o estado representado pela linha cheia, pois o tempo local de cada processo que participa do estado global representado pela linha tracejada não é maior do que o tempo local fornecido pelo outro estado global.

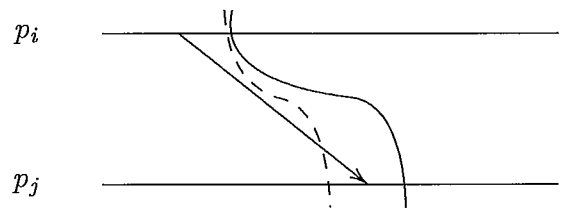

Figura 3.2: Estado global mais adiantado

Em relação à representação dos tipos de *breakpoints* tratados nesta tese, utilizaremos nos eixos de tempos locais barras mais grossas para indicarmos os períodos durante os quais os predicados locais são verdadeiros. Assim, para *breakpoints* sobre predicados disjuntivos, a linha que representa o estado global deve passar por pelo menos uma das barras grossas nos eixos de tempos, pois para um predicado disjuntivo ser satisfeito, pelo menos um dos predicados locais deve ter se tornado verdadeiro. Estes predicados locais poderiam ser, por exemplo, testes de valores de variáveis e deste modo o predicado disjuntivo poderia ser expresso pela condição "a variável x de p_i é maior do que três ou a variável y de p_k é menor do que cinco ou a variável z de p_ℓ é maior ou igual a dois". A Figura 3.3 ilustra um estado global onde o predicado disjuntivo é satisfeito quando o predicado local do processo p_i é satisfeito.

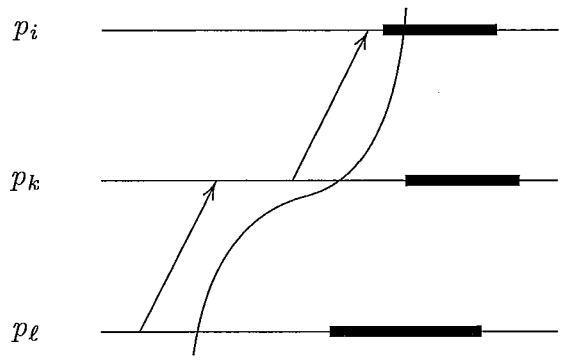

Figura 3.3: Estado global sobre predicado disjuntivo

Quando consideramos *breakpoints* incondicionais, o estado global deve coincidir com os *breakpoints* locais de cada processo. Como exemplo de *breakpoint* incondicional, poderíamos ter a seguinte condição: "o processo p_i na linha 100 e o processo p_k na linha 200 e o processo p_ℓ na linha 50", cada processo na primeira vez em que a respectiva linha foi atingida. Deste modo, na Figura 3.4 a linha que representa o estado global passa pelas marcas nos eixos de tempos que representam os *breakpoints* incondicionais locais.

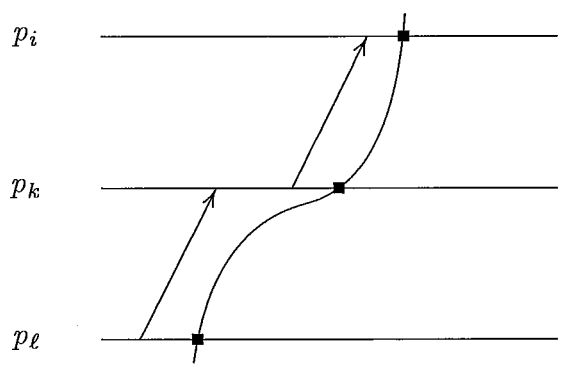

Figura 3.4: Estado global sobre *breakpoint* incondicional

Neste mesmo contexto, a Figura 3.5 mostra um estado global que satisfaz um predicado conjuntivo, onde a linha que o representa passa pelas barras grossas de todos os processos. Em relação aos predicados locais citados no caso de predicados disjuntivos, agora poderíamos ter a seguinte condição: "a variável x de p_i é maior do que três e a variável y de p_k é menor do que cinco e a variável z de p_ℓ é maior ou igual a dois". No caso de o predicado local ser estável, a barra grossa terminará na extremidade do eixo, indicando que uma vez verdadeiro o predicado estável permanece verdadeiro até o final da execução. Como exemplo de predicado local estável em um processo p_i podemos citar a condição "o arquivo F foi acessado pelo processo p_i", que será verdadeira a partir do ponto em que o arquivo F for acessado pelo processo p_i e permanecerá verdadeira depois disso.

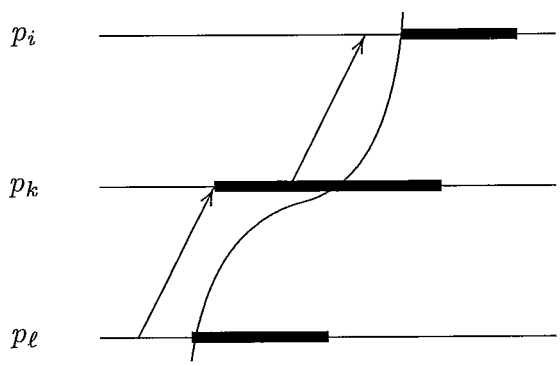

Figura 3.5: Estado global sobre predicado conjuntivo

Nestas representações, as setas indicam apenas as mensagens que a própria aplicação envia. Mensagens especiais, utilizadas pelos algoritmos projetados nesta tese, chamadas de mensagens de *broadcast*, não serão representadas nas figuras.

3.3 Análise dos Algoritmos

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.

Capítulo 4

Trabalhos Relacionados

4.1 Descrição

A habilidade de se estabelecer *breakpoints* é possivelmente a característica mais importante de um depurador. Conforme já relatado, em depuradores seqüenciais, os *breakpoints* geralmente são pontos de interesse na execução do programa onde um predicado é satisfeito. Estes predicados são expressos em termos de eventos que correspondem a um comportamento particular ou mudança no estado do programa. Em depuradores de programas distribuídos, o conjunto de predicados que podem ser usados para descrever um *breakpoint* é muito maior do que para programas seqüenciais e, normalmente, envolvem estados globais, o que se torna um problema devido à ausência de um relógio global.

Apesar de a literatura sobre depuração de programas paralelos distribuídos estar se tornando extensa, poucos autores tratam do problema de projetar algoritmos para detecção de *breakpoints* expressos por um conjunto

de condições inter-relacionadas espalhadas pelo sistema. Descreveremos, a seguir, sucintamente, os trabalhos que mais se aproximam do nosso.

Dentre os artigos mais significativos da área podemos citar o de Miller e Choi [29]. Eles apresentam um algoritmo para parar um programa distribuído em um estado consistente (chamado de algoritmo de *Halting*), que é uma extensão do algoritmo de Chandy e Lamport para gravação de estados globais [6]. Com base neste algoritmo, ainda são apresentados outros algoritmos para tratar de *breakpoints* sobre dois tipos de predicados: disjuntivo e *linked*. O primeiro é satisfeito quando pelo menos um dos predicados locais dos processos que participam na computação é satisfeito (é do mesmo tipo que o considerado nesta tese) e o algoritmo que o trata consiste simplesmente da execução do algoritmo de *Halting* pelo processo que detecta a satisfação do predicado local. O segundo é especificado por uma seqüência de eventos que podem ser ordenados pela relação "aconteceu antes" e permite uma abordagem mais ampla quando os eventos são substituídos por predicados disjuntivos. Assim, vejamos como o algoritmo funciona quando os eventos são substituídos por predicados disjuntivos. O predicado do tipo *linked* é enviado para cada processo envolvido no primeiro componente (que é um predicado disjuntivo). Quando um processo detecta que o predicado disjuntivo foi satisfeito, este processo cria um novo predicado *linked* retirando o primeiro componente. Este novo predicado é enviado para os processos envolvidos no predicado disjuntivo correspondente ao segundo componente, e assim este procedimento é repetido até que o último predicado disjuntivo no predicado *linked* seja satisfeito, quando então o processo inicia o algoritmo de *Halting*. Estes algoritmos não possuem nenhum compromisso com a detecção do *breakpoint* no estado global mais adiantado, já que uma vez que um

processo detecte a satisfação do predicado, os outros processos continuam as suas execuções até receberem a mensagem enviada a partir do processo que iniciou o *Halting*.

O problema de detecção de estados globais mais adiantados foi tratado por Manabe e Imase [24], que forneceram algoritmos distribuídos 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. Ambos os algoritmos são baseados na repetição de uma computação distribuída baseada em um histórico de execução, técnica descrita mais profundamente no Capítulo 2. No caso conjuntivo, o algoritmo se baseia em colocar os processos em estados passivos e ativos. Se um processo está sendo executado, ele é chamado de ativo e o programa está parado quando todos os processos que o compõem estão em estados passivos. Inicialmente todos os processos estão ativos e permanecem neste estado até que o predicado local seja satisfeito, quando então se tornam passivos. Entretanto, pode acontecer que um processo ativo tente receber uma mensagem que não chega porque o processo que a enviaria está em estado passivo. Neste caso, o processo receptor envia uma mensagem de controle pedindo ao processo que está em estado passivo para enviar a mensagem (consultando, para isso, o histórico de execução). Um processo, ao receber uma mensagem de controle, se torna ativo e executa até enviar a mensagem requisitada, depois disso pode se tornar passivo novamente se o predicado local estiver satisfeito. Em outras palavras, um processo está ativo quando seu predicado não estiver satisfeito ou quando ele deve enviar uma mensagem. No caso disjuntivo, eles provam que usando a mesma técnica seria impossível parar no primeiro estado global e apresentam

um algoritmo que exibe o valor de uma expressão quando o predicado disjuntivo é verdadeiro. Para isso, é usado um vetor de tempos locais que indica o estado global para o qual o predicado é satisfeito e, através da recuperação dos valores das variáveis para aqueles tempos (gravadas previamente), a expressão é calculada.

Uma solução centralizada foi mais recentemente proposta por Garg e Waldecker [11] para detectar predicados conjuntivos genéricos. Nesta solução, todos os processos são chamados de "nonchecker", exceto um deles que receberá mensagens de todos os outros e é chamado de "checker". Sempre que o predicado local de um processo é satisfeito pela primeira vez desde a última mensagem enviada, ele gera uma mensagem de depuração contendo o seu vetor de tempos e envia para o *checker*, que então testa se o predicado conjuntivo está satisfeito. O processo não precisa enviar seu vetor de tempos sempre que o predicado local é satisfeito, é necessário enviá-lo apenas depois que mensagens são enviadas. O *checker* possui uma fila separada FIFO para cada processo envolvido no predicado e mensagens de depuração que chegam destes processos são colocadas nas filas apropriadas. A tarefa do *checker* é examinar a ordenação entre os vetores. Para o predicado conjuntivo ser satisfeito, o *checker* deve encontrar um conjunto de vetores, um de cada fila, tal que cada um deles seja "incomparável" a todos os outros no conjunto, ou seja, para quaisquer dois vetores u e v, u não representa um estado global mais adiantado do que v, nem v representa um estado global mais adiantado do que u. Os autores também argumentam que esta detecção pode ser distribuída dividindo-se os processos em grupos organizados hierarquicamente.

Outros autores também têm considerado os mesmos problemas que os

tratados neste trabalho. Um exemplo é o trabalho de Spezialetti [38], que propôs uma solução para detectar *breakpoints* baseada no conceito de "regiões simultâneas" desenvolvido por Spezialetti e Kearns [39]. Como tais regiões perdem muitos dos possíveis estados globais em uma computação distribuída, a solução deles é incompleta e só pode ser aplicada a *breakpoints* sobre predicados estáveis embora sem a garantia de se encontrar os estados globais mais adiantados. Outro exemplo ainda é o apresentado por Haban e Weigel [13], onde o algoritmo de detecção de predicados globais, que são representados por uma árvore binária distribuída sobre os processos, pode também perder estados globais.

Uma outra questão relacionada foi tratada por Cooper e Marzullo [8]. Os artigos anteriores analisam uma execução em particular e não conseguem responder se existe uma execução durante a qual um estado global satisfaça um predicado. Cooper e Marzullo tratam este problema através de dois algoritmos. O primeiro é usado para detectar se existe alguma execução do programa no qual o predicado é satisfeito. O segundo detecta se, para todas as execuções do programa, o predicado é satisfeito. Ambos os algoritmos são centralizados e são baseados em "lattices" de estados globais. Entretanto, o número de estados globais cresce exponencialmente com o tamanho do sistema para algumas computações, o que torna estes algoritmos inúteis na prática.

Ainda podemos citar trabalhos que tratam de outros tipos de predicados globais, como o de Tomlinson e Garg [43] e Hurfin *et al* [17]. No primeiro, são tratados os predicados relacionais da forma $(x_0 + x_1 < C)$, onde x_0 e x_1 são valores inteiros nos processos P_0 e P_1 em um sistema de N processos, mas

o predicado é composto de apenas duas variáveis. No segundo, são tratados o que eles chamam de seqüência atômica de predicados. Este problema é o mesmo que o tratado por Miller e Choi (descrito acima) e chamado de predicados do tipo *linked*, mas acrescido de restrições de atomicidade. Ou seja, no trabalho anterior, em uma seqüência de predicados, cada um era detectado por vez sem a necessidade de descartar soluções e o comprimento dos predicados já satisfeitos era uma função não decrescente do tempo. Nas seqüências atômicas, um prefixo da seqüência de predicados pode ter sido satisfeito por uma computação em algum tempo e então invalidado pela ocorrência de um evento proibido. Assim, o trabalho detecta uma seqüência de predicados locais, enquanto outros predicados continuamente não são satisfeitos.

4.2 Comentários

Apesar de breve, a descrição feita deixa claro que existem ainda muitos problemas que merecem melhores soluções do que as já existentes. Por exemplo, os algoritmos propostos por Miller e Choi não possuem nenhum compromisso com a detecção do *breakpoint* no estado global mais adiantado. A idéia de detecção do estado global mais adiantado é uma extensão natural da idéia utilizada em programas seqüenciais, onde o depurador pára na primeira ocorrência da condição pesquisada pelo programador. Apesar de Manabe e Imase tratarem a questão de detecção no estado global mais adiantado, seus algoritmos exigem que um histórico de execução seja gerado em execução prévia, o que aumenta os requerimentos de memória do algoritmo. Como foi relatado, a geração de um histórico de execução é útil na reprodução de uma execução em particular. Entretanto, um histórico pode não apresentar todas

as possibilidades de erros do programa, exigindo que diversas gravações de históricos diferentes sejam realizadas, caso se tente analisar cada uma das possíveis execuções com erro. Neste contexto, a utilização de um histórico de execução é desnecessária e, como já dito, exige mais memória do algoritmo. Garg e Waldecker propõem basicamente uma solução centralizada para a detecção de *breakpoints*, realizada pelo processo chamado por eles de *checker*. Os demais algoritmos citados anteriormente são incompletos ou tratam de outras questões relacionadas.

Capítulo 5

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 transmiti-la 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 con-

texto).

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, 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, da mesma forma que os "vetores de tempo" tratados em [9] [26] e que representações adotadas em outros trabalhos relacionados com este nosso (mencionados no Capítulo 4). 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*. Neste caso, medidas adicionais são tomadas para parar o programa, conforme discutiremos no Capítulo 8.

Na seção 5.1 deste capítulo examinaremos o algoritmo usado para detectar *breakpoints* expressos como predicados disjuntivos, que chamaremos de DETECT_DP. Este algoritmo emprega somente mensagens de *computação*.

A fim de simplificar a compreensão dos algoritmos apresentados nos capítulos seguintes, apresentaremos também neste capítulo, seção 5.2, o algoritmo BROADCAST_WHEN_TRUE, que detecta *breakpoints* expressos como predicados conjuntivos, onde a computação propriamente não envia mensagens, ou seja, somente mensagens do tipo *broadcast* são empregadas.

Os outros algoritmos apresentados nos capítulos seguintes estão entre estes dois extremos (no que se refere aos diferentes estilos de comunicação entre nós do sistema alvo) dados pelos algoritmos DETECT_DP e BROADCAST_WHEN_TRUE.

5.1 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, como ilustrado na Figura 5.1. Assim, é possível que mais de um processo detecte a ocorrência do *breakpoint*, entretanto em estados globais diferentes.

Na Figura 5.1 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

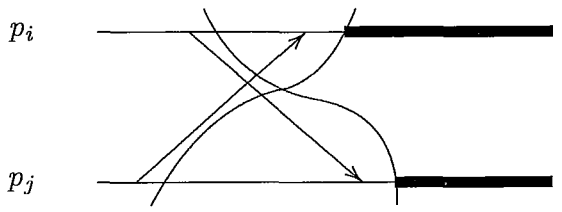

Figura 5.1: Dois estados globais mais adiantados onde o predicado disjuntivo é satisfeito

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.

Antes de proceder, são apresentados dois lemas significativos no projeto dos algoritmos deste trabalho, devido às modificações que o vetor gs_i sofre nestes algoritmos.

Lema 1. Para todo $p_i \in N$, se gs_i é um estado global tal que $gs_i[i] < lt_i$, e nenhuma mensagem é recebida por p_i no tempo t tal que $gs_i[i] < t \leq lt_i$, então $gs_i(lt_i)$ é também um estado global.

Prova: Se $gs_i(lt_i)$ não é um estado global, então devem existir p_k, $p_\ell \in N$ de forma que uma mensagem de computação foi enviada por p_k imediatamente depois de $gs_i(lt_i)[k]$ e recebida em p_ℓ antes do que (ou em) $gs_i(lt_i)[\ell]$.

Pela definição de $gs_i(lt_i)$, e pela hipótese, temos que a mensagem deve ter sido enviada mais tarde do que $gs_i[k]$ e chegado em p_ℓ antes do que (ou em) $gs_i[\ell]$, e então gs_i não é um estado global, o que é uma contradição.

■

Lema 2. Se φ e φ' são estados globais, então o estado global resultante dos máximos dos componentes dos dois vetores é também um estado global.

Prova: Seja φ'' o máximo obtido entre φ e φ', e suponha que este não seja um estado global. Então devem existir $p_k, p_\ell \in N$ de forma que uma mensagem foi enviada por p_k imediatamente depois que $\varphi''[k]$ e recebida em p_ℓ antes do que (ou em) $\varphi''[\ell]$. Como $\varphi''[k] \geq \varphi[k]$ e $\varphi''[k] \geq \varphi'[k]$, então φ não é um estado global se $\varphi''[\ell] = \varphi[\ell]$. Da mesma forma, se $\varphi''[\ell] = \varphi'[\ell]$, então φ' não é um estado global. Ambos os casos são contradições.

■

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 pelos Lemas 1 e 2. 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.

5.1.1 O Algoritmo Detect_DP

A seguir é apresentado o algoritmo para detecção de predicados disjuntivos, com um exemplo mostrando o seu funcionamento.

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 $p_j \in N_i$:

$found\_elsewhere_i := bit_j$;
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;

Como exemplo das ações citadas acima, observemos a Figura 5.1, onde associados aos processos p_i e p_j estão predicados locais. Antes de o predicado local em q_i se tornar verdade, este executa a ação (2), enviando uma mensagem de computação a q_j e, ao receber uma mensagem de computação de q_j, executa a ação (3). Nesta última ação, $gs_i[j]$ é atualizado com o tempo em que a mensagem foi enviada por p_j. Quando lp_i se torna **true**, q_i executa a ação (1). Como a mensagem recebida anteriormente de q_j não trazia indicação (bit_j) de que q_j já tinha detectado o predicado disjuntivo, q_i atualiza $gs_i[i]$ e $found_i$, detectando, assim, o estado global que satisfaz o predicado disjuntivo, representado pelo vetor gs_i. As mesmas ações são executadas por q_j que detecta o outro estado global mais adiantado que satisfaz o predicado disjuntivo.

5.1.2 Corretude e Complexidade

Os teoremas 3 e 4 dados a seguir estabelecem, respectivamente, a corretude e a complexidade do algoritmo DETECT_DP.

Teorema 3. Existem um $p_i \in N$ e um $t \geq 0$ tais que as três condições a seguir são equivalentes entre si para o algoritmo DETECT_DP.

(a) Existe um estado global φ tal que $lp_k =$ **true** no tempo $\varphi[k]$ para no mínimo um $p_k \in N$;

(b) $found_i$ se torna **true** no tempo $lt_i = t$;

(c) No tempo $lt_i = t$, gs_i é o estado global mais adiantado no qual $lp_k =$ **true** para no mínimo um $p_k \in N$.

Prova:

(a) $\Rightarrow$ (b):

Pelo menos um processo $p_k \in N$ para o qual $lp_k =$ **true** em $\varphi[k]$ deve ter atingido este estado quando $found\_elsewhere_k =$ **false**. Esta afirmação é consequência imediata da ação (1), com p_i sendo este processo particular e t sendo o tempo local no qual lp_i se torna **true**.

(b) $\Rightarrow$ (c):

Pela hipótese e ação (1), $found\_elsewhere_i$ só pode ter se tornado **true** após o tempo t. Pelos Lemas 1 e 2, o gs_i produzido pela ação (1), o $gs_i(lt_i)$ usado na ação 2, e o gs_i produzido pela ação (3) devem todos ser estados globais. Como conseqüência disso, pela ação (1) gs_i é no tempo t um estado global no qual $lp_i =$ **true**. Se gs_i não fosse um estado global mais adiantado no qual $lp_k =$ **true** para no mínimo um $p_k \in N$, então ou $found\_elsewhere_i$ teria se tornado **true** pelas ações (2) e (3) antes de t, e então $found_i$ seria **false** em t, o que é uma contradição, ou lp_k seria **true** para algum $p_k \in N$ desde o início, o que é rejeitado pela hipótese sobre os valores iniciais destas

variáveis.

(c) $\Rightarrow$ (a):

Esta etapa da prova é imediata.

■

Teorema 4. O algoritmo DETECT_DP tem 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.

Prova: Cada uma das $O(c)$ mensagens de computação carrega um vetor de n componentes representando tempos locais, onde cada um dos componentes é um inteiro não maior do que T, conforme consideramos. Logo, a complexidade de mensagens é $O(cn\log T)$ *bits*. Como somente mensagens de computação são empregadas, a complexidade de tempo global é $O(1)$. Para cada mensagem recebida, os n elementos do vetor gs são testados e podem ser atualizados. Assim, a complexidade de tempo local é $O(n)$ por mensagem recebida. Para $p_i \in N$, o processo q_i precisa armazenar o vetor gs_i, que requer $O(n\log T)$ *bits*.

■

5.2 Propagação de Informação Global

A detecção dos outros tipos de *breakpoints* considerados nesta tese é uma tarefa mais difícil em comparação com a detecção de *breakpoints* sobre predi-

cados disjuntivos. Estes outros casos incluem os *breakpoints* incondicionais, os *breakpoints* sobre predicados conjuntivos (tanto no caso estável como no geral) e 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.

5.2.1 O Algoritmo Broadcast_when_true

Apresentamos nesta seção o algoritmo BROADCAST_WHEN_TRUE e um exemplo de sua execução.

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 $gc_i[1] \wedge \cdots \wedge gc_i[n]$ **then**

$found_i :=$ **true**

else

Envie (gc_i, gs_i) para todo q_k tal que $p_k \in N_i$;

(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 $gc_i[1] \wedge \cdots \wedge gc_i[n]$ **then**

$found_i := \mathbf{true}$

else

Envie (gc_i, gs_i) para todo q_k tal que $p_k \in N_i$

end;

Na Figura 5.2, suponha que exista um canal conectando cada par de processos. O processo q_i, ao detectar que lc_i se tornou **true**, executa a ação (1) e, caso não tenha ainda recebido as mensagens de *broadcast* de p_k e p_ℓ, inicia um novo *broadcast* com os vetores gs_i e gc_i atualizados. Ao receber uma mensagem de *broadcast*, q_i executa a ação (2), atualizando os mesmos vetores e prosseguindo com o *broadcast*, se lc_i é **true**. Caso esta mensagem de *broadcast* carregue valores iguais aos já recebidos em alguma mensagem de *broadcast* anterior, não ocorrem atualizações nos vetores gc_i e gs_i, $changed_i$ permanece **false** e o *broadcast*, neste caso, é descontinuado por q_i.

Na nossa figura um processo pode receber até duas mensagens de *broadcast* iguais. Por exemplo digamos que q_k execute a ação (1), este envia mensagens de *broadcast* para q_i e q_ℓ, que por sua vez podem prosseguir com o *broadcast*. Caso $gs_\ell[\ell]$ seja **true**, q_ℓ envia a mensagem recebida para q_i e q_k, mas neste caso esta mensagem de *broadcast* não provoca mudanças nos vetores destes processos, que, assim, descontinuam este *broadcast*. O *broadcast* também é descontinuado caso este atinja q_i quando lc_i é **false**.

A detecção da satisfação do predicado conjuntivo estável na ausência de mensagens de computação ocorre quando um processo $q_j \in N$ detecta que $gc_j[k] = \mathbf{true}$ para todo $p_k \in N$, através da ação (1) ou (2).

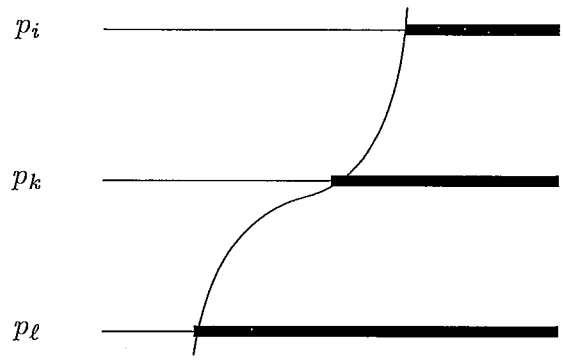

Figura 5.2: Estado global sobre predicado conjuntivo estável na ausência de mensagens de computação

5.2.2 Corretude e Complexidade

Os dois teoremas a seguir são relacionados às propriedades do algoritmo BROADCAST_WHEN_TRUE.

Teorema 5. Existem um $p_i \in N$ e um $t \geq 0$ tais que as três condições a seguir são equivalentes entre si para o algoritmo BROADCAST_WHEN_TRUE.

(a) Existe um estado global φ tal que $lc_k =$ **true** no tempo $\varphi[k]$ para todo $p_k \in N$;

(b) $found_i$ se torna **true** no tempo $lt_i = t$;

(c) No tempo $lt_i = t$, gs_i é o estado global mais adiantado no qual $lc_k =$ **true** para todo $p_k \in N$.

Prova:

(a) $\Rightarrow$ (b):

Se exatamente um processo participa na condição global, então pela ação

(1), $found_i$ se torna **true**, com $p_i \in N$ sendo este processo e t o tempo no qual lc_i se torna **true**. Nenhuma mensagem é enviada neste caso. Se no mínimo dois processos participam, então no mínimo um deles, digamos $p_k \in N$, é tal que q_k, pela ação (1), envia uma mensagem de *broadcast* para seus vizinhos quando lc_k se torna **true**, que pela ação (2) passam a informação adiante desde que a atualização tenha introduzido mudanças e suas condições locais sejam satisfeitas também. Como este *broadcast* carrega lc_k, ele deve introduzir mudanças ao atingir todo nó pela primeira vez e por isso é propagado. Isto acontece à condição local de todo nó que participa do *breakpoint*, e então pelo menos um processo, digamos q_i, ao ser atingido pelos *broadcasts* e tendo $lc_i =$ **true**, atualiza $found_i =$ **true**. O valor de t aqui é o tempo em que o último *broadcast* atinge q_i pela ação (2) ou o tempo no qual lc_i se torna **true** pela ação (1).

(b) $\Rightarrow$ (c):

Pelos Lemas 1 e 2, o gs_i produzido nas ações (1) e (2) são estados globais. Conseqüentemente, e pelas ações (1) e (2) também, no tempo t gs_i é um estado global no qual $lc_k =$ **true** para todo $p_k \in N$. Por causa da ausência de mensagens de *computação* é imediato que gs_i seja o estado global mais adiantado, o que implica que $gs_i[k]$ é ou zero ou o tempo no qual lc_k se torna **true**.

(c) $\Rightarrow$ (a):

Esta parte da prova é imediata.

■

Teorema 6. O algoritmo BROADCAST_WHEN_TRUE possui complexidade de mensagens de $O(n^2 e \log T)$ *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.

Prova: O pior caso é aquele em que todos os nós iniciam o algoritmo concorrentemente e o *broadcast* iniciado por um nó passa por todos os canais. Como as mensagens são enviadas com dois vetores de n componentes, um com componentes de um único *bit* e o outro com inteiros limitados por T, a complexidade de mensagens se torna $O(n^2 e \log T)$ *bits*. A complexidade de tempo global é resultado do fato de que nenhuma cadeia causal de mensagens engloba mais do que $O(n)$ mensagens, o que é necessário para um *broadcast* atingir todos os nós. A complexidade de tempo local e as exigências de memória são as mesmas do algoritmo DETECT_DP, por isso dadas pelo Teorema 4, embora no caso da memória este algoritmo também utilize o vetor gc, o que não altera a complexidade.

■

5.3 Comentários

Como vimos, os algoritmos DETECT_DP e BROADCAST_WHEN_TRUE se situam entre dois extremos, já que no primeiro caso somente mensagens de *computação* são empregadas, enquanto no último caso somente mensagens de *broadcast* são necessárias. Outras situações entre estes extremos são examinadas a seguir, e então as mensagens envolvidas se tornam do tipo $(computation, gc_i, gs_i, body)$ para mensagens de computação enviadas por q_i,

e $(broadcast, gc_i, gs_i, \mathbf{nil})$ para mensagens de *broadcast* enviadas por q_i. O campo **nil** somente é usado para produzir mensagens com o mesmo número de campos, com o primeiro campo sendo usado para diferenciar os tipos de mensagens.

Capítulo 6

Breakpoints Incondicionais

Neste capítulo 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.

O algoritmo DETECT_UBP pode ser considerado uma combinação dos algoritmos DETECT_DP e BROADCAST_WHEN_TRUE, discutidos no capítulo 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 capítulo 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 lu-

gar 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

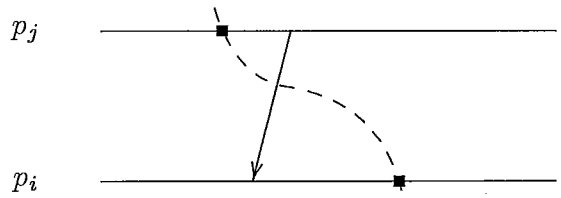

Figura 6.1: Erro na colocação dos *breakpoints* incondicionais locais

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 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 6.1, 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 men-

sagens 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 6.2, 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.

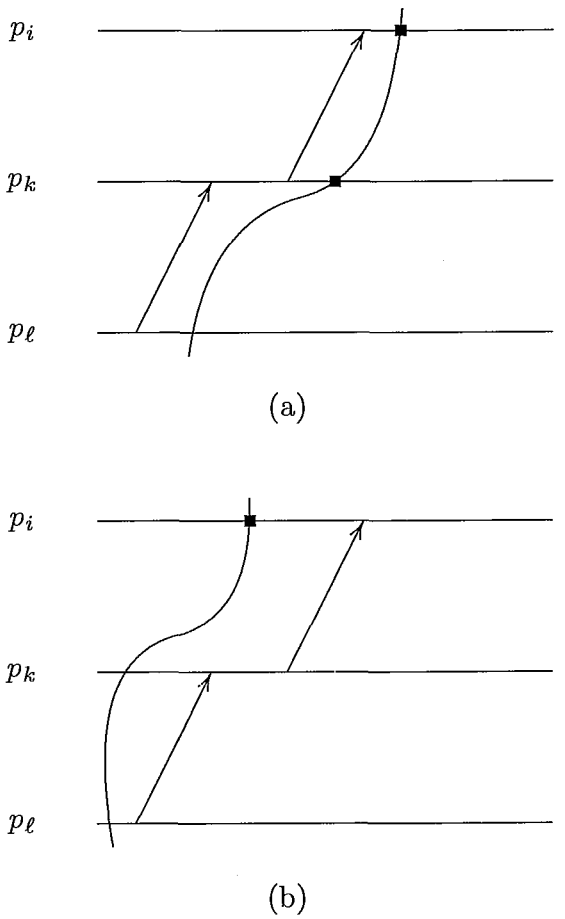

Figura 6.2: Estados globais mais adiantados e *breakpoints* incondicionais

6.1 O Algoritmo Detect_UBP

A seguir são apresentadas as ações que compõem o algoritmo DETECT_UBP e um exemplo mostrando a sua utilização.

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 $ub_i[k] \neq$ **false** for all $k = 1, \ldots, n$ **then**
$found_i :=$ **true**
else
Envie $(broadcast, ub_i, gs_i,$ **nil**$)$ para todo q_k tal que $p_k \in N_i$
end;

(2) Ao receber $(broadcast, ub_j, gs_j,$ **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 $(lub_i \neq$ **false**$)$ **and** $changed_i$ **then**

if $ub_i[k] \neq$ **false** para todo $k = 1, \ldots, n$ **then**

$found_i :=$ **true**

else

Envie $(broadcast, ub_i, gs_i,$ **nil**$)$ para todo q_k tal que $p_k \in N_i$

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;

Tendo apresentado o algoritmo DETECT_UBP, observemos a parte (a) da Figura 6.2, onde, associados aos processos p_i e p_k, existem *breakpoints* incondicionais locais. Antes que p_i atinja seu *breakpoint* local, ele receberá uma mensagem de *computação* de q_k. Quando q_i atingir o *breakpoint* local e executar a ação (1), este poderá já ter recebido a mensagem de *broadcast* de q_k pela ação (2) e, neste caso, detectará a satisfação do *breakpoint* incondicional, pois $ub_i[k] \neq$ **false** para todo $p_k \in N$, atualizando $found_i$ com **true**. Caso contrário, ou seja, q_i não tenha recebido a mensagem de *broadcast* de q_k, q_i iniciará um *broadcast* e q_k ou q_ℓ poderá detectar a satisfação do *breakpoint* incondicional, uma vez que p_k tenha atingido seu *breakpoint* local. Antes de q_k atingir seu *breakpoint* local, este receberá uma mensagem de *computação* de q_ℓ, executando a ação (4). Nesta ação ocorrerá a atualização de $gs_k[\ell]$ com o tempo relativo ao envio da mensagem de p_ℓ para p_k.

Caso o *breakpoint* local de p_k estivesse localizado antes do envio da men-

sagem para p_i, pela ação (4), quando p_i recebesse a mensagem de *computação* de q_k (com $ub_k[k]$ atualizado com **true** pela ação (1)), in_error_i seria atualizado com **true**, indicando erro na colocação dos *breakpoints*, que não constituiriam um estado global.

Na parte (b) da Figura 6.2, apenas p_i possui um *breakpoint* local associado. Quando q_i atingir seu *breakpoint* local, ele executará a ação (1) atualizando $found_i$ com **true**, pois $ub_i[k] \neq$ **false** para todo $p_k \in N$. A cadeia de mensagens de *computação* iniciada por q_ℓ, ao atingir q_i (ação 4), não atualizará os vetores pois $found_i$ já será **true**.

6.2 Corretude e Complexidade

A seguir são apresentadas as propriedades do algoritmo DETECT_UBP relacionadas à sua corretude e complexidade.

Teorema 7. Existem um $p_i \in N$ e um $t \geq 0$ tais que as quatro condições a seguir são equivalentes entre si para o algoritmo DETECT_UBP.

(a) Existe um estado global φ tal que $\varphi[k] = lub_k$ para todo $p_k \in N$ tal que $lub_k < \infty$;

(b) in_error_k nunca se torna **true** para qualquer $p_k \in N$;

(c) $found_i$ se torna **true** no tempo $lt_i = t$;

(d) No tempo $lt_i = t$, gs_i é o estado global mais adiantado no qual $gs_i[k] = lub_k$ para todo $p_k \in N$ tal que $lub_k < \infty$.

Prova:

(a) $\Rightarrow$ (b):

Suponha que exista $p_k \in N$ tal que in_error_k se torne **true**. Pela ação (4), isto deve acontecer no recebimento, quando $ub_k[k] =$ **false**, de uma mensagem de *computação* que faz parte de uma cadeia causal de mensagens de *computação* iniciada em, digamos q_ℓ tal que $p_\ell \in N_k$, enviada quando $ub_\ell[\ell] =$ **true**. Nenhum vetor φ tal que $\varphi[k] = lub_k$ e $\varphi[\ell] = lub_\ell$ pode então ser um estado global, e porque $lub_k < \infty$ e $lub_\ell < \infty$, tem-se uma contradição.

(b) $\Rightarrow$ (c):

Se in_error_k nunca se torna **true** para qualquer $p_k \in N$, então as ações (1) e (2), em relação às mensagens de *broadcast*, são idênticas às ações (1) e (2), respectivamente, do algoritmo BROADCAST_WHEN_TRUE. Esta parte da prova é análoga à parte (a) $\Rightarrow$ (b) da prova do Teorema 5.

(c) $\Rightarrow$ (d):

Pelos Lemas 1 e 2, o gs_i produzido pela ação (1), o $gs_i(lt_i)$ e $alt\_gs_i(lt_i)$ usados na ação (3), e o gs_i e alt_gs_i produzidos pelas ações (2) e (4) são todos estados globais. Isto implica, pelas ações (1) e (2) e no tempo t, que gs_i é um estado global no qual $ub_i[k] \neq$ **false** para todo $p_k \in N$, ou, equivalentemente, um estado global tal que $gs_i[k] = lub_k$ para todo $p_k \in N$ tal que $lub_k < \infty$. A fim de mostrar que gs_i é o estado global mais adiantado com estas características, considere qualquer outro vetor de n componentes de tempos locais, chamado φ, tal que $\varphi[k] = gs_i[k]$ para todo $p_k \in N$ tal que $lub_k < \infty$, e $\varphi[k] < gs_i[k]$ para pelo menos um $p_k \in N$ tal que $lub_k = \infty$. Em relação a este particular p_k, para que $gs_i[k]$ tenha sido atualizado com

um valor maior do que $\varphi[k]$, uma cadeia causal de mensagens de *computação* deve ter existido de p_k (partindo no tempo $gs_i[k]$) para algum $p_\ell \in N$, onde pela ação (4) deve ter chegado em q_ℓ quando $ub_\ell[\ell] =$ **false** (caso contrário gs_ℓ não teria sido atualizado e também não gs_i através do *broadcast*). Além disso, como in_error_ℓ deve ter permanecido **false**, todo processo envolvido nesta cadeia (exceto q_ℓ mas incluindo q_k) deve ter tido um **undefined** no seu *breakpoint* incondicional local (para q_k, $ub_k[k] =$ **undefined**). Mas, como $ub_\ell[\ell]$ foi encontrado **false**, φ não pode ser um estado global tal que $\varphi_k = lub_k$ para todo $p_k \in N$ tal que $lub_k < \infty$.

(d) $\Rightarrow$ (a):

É imediato.

■

Teorema 8. O algoritmo DETECT_UBP possui complexidade de mensagens de $O((c+ne)n\log T)$ *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.

Prova:

A complexidade de mensagens deste algoritmo é a soma das complexidades de mensagens do algoritmo DETECT_DP e o algoritmo BROADCAST_WHEN_TRUE. Pelos Teoremas 4 e 6, obtém-se $O((c+ne)n\log T)$ *bits*. As complexidades restantes e necessidade de memória são exatamente as mesmas do algoritmo BROADCAST_WHEN_TRUE e, portanto são dadas como no Teorema 6, embora os processos que não participam do *breakpoint*

incondicional possuam adicionalmente um vetor de n componentes com uma visão alternativa do estado global, o que não altera as complexidades.

■

Capítulo 7

Breakpoints sobre Predicados Conjuntivos

Neste capítulo 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.

7.1 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* (veja a Figura 6.1). 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. A Figura 7.1 ilustra esta situação. Portanto, o algoritmo apresentado nesta seção pode ser considerado uma simplificação do algoritmo DETECT_UBP, onde as condições de erro não precisam ser tratadas.

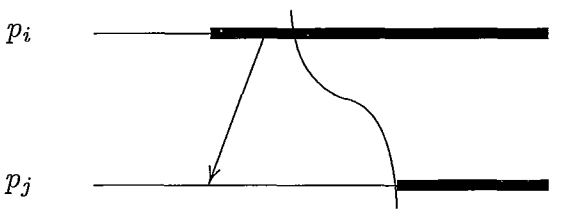

Figura 7.1: Estado global e *breakpoint* conjuntivo estável

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 príncipios 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 7.2, 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

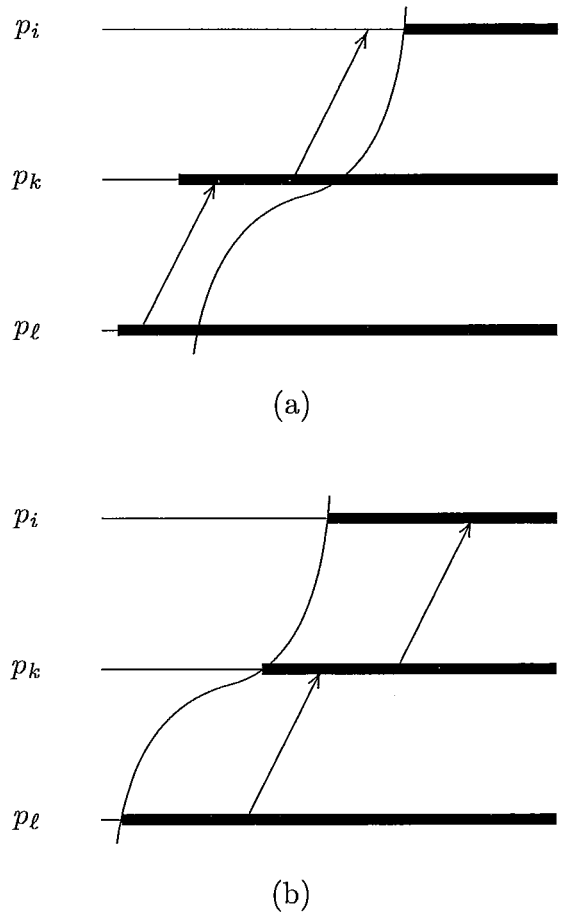

Figura 7.2: Estados globais mais adiantados e *breakpoints* sobre predicados conjuntivos estáveis

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 $p_k \in N$.

7.1.1 O Algoritmo Detect_stable_CP

São mostrados a seguir o algoritmo DETECT_STABLE_CP e um exemplo.

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 $cp_i[1] \wedge \cdots \wedge cp_i[n]$ **then**
  $found_i :=$ **true**
else
  Envie $(broadcast, cp_i, gs_i, \textbf{nil})$ para todo q_k tal que $p_k \in N_i$;

(2) Ao receber $(broadcast, cp_j, gs_j, \textbf{nil})$ 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
          $gs_i[k] := gs_j[k]$;
          $cp_i[k] := cp_j[k]$;
          $changed_i :=$ **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 $cp_i[1] \wedge \cdots \wedge cp_i[n]$ **then**

$found_i :=$ **true**

else

Envie $(broadcast, cp_i, gs_i, \mathbf{nil})$ para todo q_k tal que $p_k \in N_i$

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;

Apresentado o algoritmo, voltemos à parte (a) da Figura 7.2, onde existe um predicado local estável associado a cada processo p_i, p_k e p_ℓ. Nesta figura, q_i, antes de atualizar lp_i com **true**, executa a ação (4) ao receber a mensagem de computação de q_k, atualizando os elementos $alt\_gs_i[k]$ e $alt\_gs_i[\ell]$. O primeiro com o tempo relativo ao envio da mensagem de computação de p_k

para p_i e o segundo com o tempo de envio da mensagem de computação de p_ℓ para p_k, pois, obrigatoriamente, antes de q_k enviar a mensagem para q_i, este recebeu uma mensagem de computação de q_ℓ, que modificou $alt\_gs_k[\ell]$.

As mensagens de *broadcast* de q_k e q_ℓ, ao atingirem q_i, poderão atualizar os vetores cp_i, gs_i e alt_gs_i pela ação (2). Assim, p_i ao detectar que lp_i se tornou **true** (ação (1)), já tendo atualizado $cp_i[k]$ e $cp_i[\ell]$, atualizará os vetores cp_i, gs_i, alt_gs_i e detectará a satisfação do predicado conjuntivo estável.

Em relação à parte (b) da Figura 7.2, o processo q_i, ao detectar que lp_i se tornou **true** na ação (1), atualiza os vetores alt_gs_i, gs_i e cp_i e, caso tenha recebido as mensagens de *broadcast* de q_k e q_ℓ, detecta a satisfação do predicado ($found_i$ = **true**). Caso contrário, q_i inicia um *broadcast*. O mesmo ocorre com os processos q_k e q_ℓ. Cada um dos processos q_i, q_k e q_ℓ, ao receber mensagens de *broadcast* (ação 2) atualiza os vetores e testa se o predicado é satisfeito. No recebimento de mensagens de *computação*, os processos já terão executado a ação (1) e, portanto, estas não influenciarão o estado global detectado.

7.1.2 Corretude e Complexidade

As propriedades de corretude e complexidade do algoritmo DETECT_STABLE_CP são estabelecidas nos seguintes teoremas.

Teorema 9. Existem um $p_i \in N$ e um $t \geq 0$ tais que as seguintes três condições são equivalentes entre si para o algoritmo DETECT_STABLE_CP.

(a) Existe um estado global φ tal que lp_k = **true** no tempo $\varphi[k]$ para todo $p_k \in N$;

(b) $found_i$ se torna **true** no tempo $lt_i = t$;

(c) No tempo $lt_i = t$, gs_i é o estado global mais adiantado no qual $lp_k =$ **true** para todo $p_k \in N$.

Prova:

(a) $\Rightarrow$ (b):

Se exatamente um processo participa da condição global, então pela ação (1), $found_i$ se torna **true** com $p_i \in N$ sendo este processo e t o tempo no qual lp_i se torna **true**. Nenhuma mensagem de *broadcast* é enviada neste caso. Se no mínimo dois processos participam, então pelo menos um deles, digamos $p_k \in N$, é tal que q_k, pela ação (1), envia uma mensagem de *broadcast* para seus vizinhos quando lp_k se torna **true**. Um processo, por exemplo q_i, só não continua a propagação do *broadcast* ao receber esta mensagem, pela ação(2), caso ou lp_i seja **false** ou $changed_i$ seja **false**. O primeiro caso é tratado da mesma forma que no algoritmo BROADCAST_WHEN_TRUE. Esta parte da prova então segue as mesmas linhas da parte (a) $\Rightarrow$ (b) na prova do Teorema 5. O segundo caso ocorre quando gs_i não é modificado pela mensagem de *broadcast* recebida, ou seja, gs_i já tinha sido atualizado. Esta atualização pode ter acontecido através das ações (1) ou (2). Se gs_i foi atualizado na ação (2) anteriormente, significa que esta informação já tinha sido propagada por uma mensagem de *broadcast* que poderia atingir todos os nós do sistema, conforme ocorria com o algoritmo BROADCAST_WHEN_TRUE. No caso de gs_i ter sido atualizado na ação (1), isto significa que esta informação estava registrada em alt_gs_i. A atualização de alt_gs_i só pode ter ocorrido nas ações (2) ou (4). Considerando a ação (2), voltamos ao caso já relatado, onde gs_i

também foi atualizado no recebimento de uma mensagem de *broadcast*. Em relação a ação (4), uma mensagem de *computação* contendo as informações carregadas pela mensagem de *broadcast* foi recebida por q_i antes de lp_i se tornar **true**. Neste caso, a propagação da informação de que lp_k = **true** é realizada através do *broadcast* iniciado por q_i.

Desta forma, como a propagação sempre ocorre, pelo menos um processo, digamos q_j, ao ser atingido pelos *broadcasts* e tendo lp_j = **true**, atualiza $found_j$ com **true**. O valor de t aqui é o tempo em que um *broadcast* atinge q_j pela ação (2) ou o tempo no qual lp_j se torna **true**.

(b) $\Rightarrow$ (c):

Pelos Lemas 1 e 2, os vetores gs_i e alt_gs_i produzidos pelas ações (1) e (2), o $alt\_gs_i(lt_i)$ usado na ação (3), e o alt_gs_i produzido pela ação (4) são todos estados globais. Uma conseqüência disto é que, pelas ações (1) e (2), gs_i é no tempo t um estado global no qual $cp_i[k]$ = **true** para todo $p_k \in N$. Para mostrar que gs_i é o estado global mais adiantado, consideremos qualquer outro vetor de n componentes de tempos locais, chamado φ, tal que lp_k = **true** no tempo $\varphi[k]$ para todo $p_k \in N$ e tal que $\varphi[k] < gs_i[k]$ para no mínimo um $p_k \in N$. Para este particular p_k, $gs_i[k]$ só pode ter recebido um valor maior do que $\varphi[k]$ se existisse uma cadeia causal de mensagens de *computação* a partir de p_k (partindo no tempo $gs_i[k]$) para algum $p_\ell \in N$, que, pela ação (1) deve ter chegado em q_ℓ quando $cp_\ell[\ell]$ = **false** (de outra forma gs_ℓ não teria sido atualizado, e nem gs_i por meio do *broadcast*). Mas, porque $cp_\ell[\ell]$ foi encontrado com o valor **false**, φ não é um estado global tal que lp_k = **true** para todo $p_k \in N$.

(c) $\Rightarrow$ (a):

Esta parte da prova é imediata.

■

Teorema 10. O algoritmo DETECT_STABLE_CP possui complexidade de mensagens de $O((c+ne)n\log T)$ 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.

Prova: As complexidades e memória requisitada para este algoritmo são as mesmas que as do algoritmo DETECT_UBP, por isso dadas como no Teorema 8.

■

7.2 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 e falso ao longo da computação diversas vezes. Esta generalidade aparentemente agrava as dificuldades do problema, mas técnicas bastante similares às adotadas no Capítulo 6 e na seção anterior 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

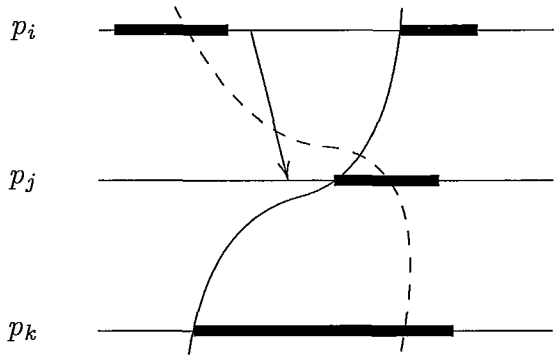

Figura 7.3: Estados globais e *breakpoints* sobre predicados conjuntivos genéricos

ser usado quando o vetor totalmente atualizado alt_gs_i não fosse necessário. Neste capítulo 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 7.3, 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 lo-

cais. 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 do Capítulo 5. 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 $alt\_cp_i[k]$.

Um outro aspecto relevante do algoritmo é que a atualização dos vetores cp e gs só ocorre no caso de recebimento de mensagem de *broadcast*. A não atualização de $cp_i[i]$, $gs_i[i]$ quando lp_i se torna **true**, mesmo quando $cp_i[i]$ é **false** (como ocorre em $alt\_cp_i[i]$) é fundamental para não se perder es-

7.2.1 O Algoritmo Detect_CP

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 $(broadcast_i, alt\_cp_i, alt\_gs_i, \textbf{nil})$ para todo q_k tal que $p_k \in N_i$
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 $(broadcast_\ell, cp_\ell, gs_\ell, \textbf{nil})$ de q_j tal que $p_j \in N_i$:

if not $found_i$ **then**
begin
$changed_i :=$ **false**;
if not $cp_i[\ell]$ **then**
for $k := 1$ **to** n **do**

if $gs_i[k] < gs_\ell[k]$ **then**

begin

$gs_i[k] := gs_\ell[k]$;

$cp_i[k] := cp_\ell[k]$;

$changed_i :=$ **true**

end;

if not $alt\_cp_i[\ell]$ **then**

for $k := 1$ **to** n **do**

if $alt\_gs_i[k] < gs_\ell[k]$ **then**

begin

$alt\_gs_i[k] := gs_\ell[k]$;

$alt\_cp_i[k] := cp_\ell[k]$;

$changed_i :=$ **true**

end;

if $changed_i$ **then**

if $cp_i[1] \wedge \cdots \wedge cp_i[n]$ **then**

$found_i :=$ **true**

else

Envie $(broadcast_\ell, cp_\ell, gs_\ell,$ **nil**$)$ para todo q_k tal que $p_k \in N_i$

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;

Antes de apresentarmos os teoremas relativos a este algoritmo, na próxima seção, vejamos um exemplo. Observemos a Figura 7.3, onde associado a cada processo existe um predicado local instável. Quando q_i detecta que lp_i se tornou **true**, este executa a ação (1), atualizando os vetores alt_cp_i, alt_gs_i e iniciando um *broadcast*. Mais tarde, ao detectar que lp_i se tornou **false**, este executa a ação (2), onde alt_cp_i e alt_gs_i são atualizados, e, ao enviar a mensagem de computação para q_j, este executa a ação (4). Antes de lp_i se tornar **true** novamente, q_i poderá receber mensagens do tipo $broadcast_i$, $broadcast_j$ e $broadcast_k$. Cada uma destas mensagens poderá atualizar, através da ação (3), os vetores gs_i, cp_i, alt_cp_i e alt_gs_i. Entretanto, a detecção do predicado conjuntivo estável não ocorrerá antes de lp_i se tornar **true** novamente, pois mesmo que os outros processos tenham seus predicados locais verdadeiros, através da mensagem $broadcast_j$, teremos associado ao predicado local do processo p_i **false**, devido à mensagem de computação recebida por q_j, onde $alt\_cp_i[i]$ era **false**. Deste modo, apenas quando lp_i se tornar **true** novamente, o predicado conjuntivo poderá ser detectado, através de um novo $broadcast_i$.

Caso as mensagens do segundo $broadcast_i$, provenientes de q_k e q_j, atinjam q_i antes que ele receba a mensagem de $broadcast_j$, estas não provocarão nenhuma modificação nos vetores de q_i (ação 3), pois $cp_i[i]$ já será **true**, devido

ao $broadcast_i$ anterior. As atualizações só ocorrerão, então, no recebimento da mensagem $broadcast_j$, que neste caso carregará informações relativas ao segundo $broadcast_i$.

7.2.2 Corretude e Complexidade

Os Teoremas 11 e 12 a seguir se referem às propriedades do algoritmo DETECT_CP. Note que P representa o número máximo de vezes que qualquer predicado local se torna verdadeiro.

Teorema 11. Existem um $p_i \in N$ e um $t \geq 0$ tais que as seguintes três condições são equivalentes entre si para o algoritmo DETECT_CP.

(a) Existe um estado global φ tal que $lp_k =$ **true** no tempo $\varphi[k]$ para todo $p_k \in N$;

(b) $found_i$ se torna **true** no tempo $lt_i = t$;

(c) No tempo $lt_i = t$, gs_i é o estado global mais adiantado no qual $lp_k =$ **true** para todo $p_k \in N$.

Prova:

(a) $\Rightarrow$ (b):

Seja φ' o estado global mais adiantado tal que $lp_k =$ **true** para todo $p_k \in N$. Se exatamente um processo participa no *breakpoint*, então pelas ações (1) e (3), facilmente se pode concluir que p_i é qualquer um dos vizinhos do processo e t é o tempo no qual o *broadcast* o atingiu. Se mais de um processo participa no *breakpoint*, então existem dois casos principais a serem

considerados.

No primeiro caso, em φ', e para todo processo que participa do *breakpoint* $p_k \in N$, lp_k se tornou **true** exatamente uma vez. Se lp_k nunca se tornou **true** novamente para qualquer um destes $p_k \in N$, então as ações (1) e (3) asseguram que pelo menos um processo $p_i \in N$ seja atingido por todos os *broadcasts* e atualize $found_i$ com **true** ao ser atingido, no tempo t, pelo último *broadcast*. Se, por outro lado, pelo menos um destes processos $p_k \in N$ for tal que lp_k se torne **true** pelo menos duas vezes, então nenhum $p_i \in N$ possui $cp_i[k] =$ **false** ao ser atingido pelos *broadcasts* posteriores ao primeiro iniciado por q_k, e então pelas ações (1) e (3), como anteriormente, pelo menos um processo $p_i \in N$ atualiza $found_i$ com **true** no tempo t. Para entender porque $cp_i[k]$ deve ser **true** quando (e se) q_i é atingido pelos *broadcasts* posteriores ao primeiro iniciado por q_k, suponha que $cp_i[k]$ estava **false** quando q_i foi atingido por qualquer um destes *broadcasts*. Pelas ações (1), (2), (3) e (5), $cp_i[k]$ só pode ter se tornado **false** no recebimento de uma mensagem de *broadcast*, cujo envio deve ter sido influenciado pelo recebimento de uma cadeia causal de mensagens de *computação* começando em p_k carregando o valor **false**. Como os canais são FIFO, esta cadeia deve ter partido de p_k depois que q_k iniciou seu primeiro *broadcast*, caso contrário o *broadcast* que atualizou $cp_i[k]$ com **false** não teria prevalescido. Mas, se este *broadcast* foi iniciado em, digamos q_ℓ para algum $p_\ell \in N$, então q_i deve ter tido $cp_i[\ell] =$ **false** ao ser atingido por este, e então ou este era o primeiro *broadcast* iniciado por q_ℓ ou o argumento inteiro deve ser repetido com p_ℓ em lugar de p_k. Eventualmente, neste argumento nós terminaríamos em uma cadeia causal de mensagens de *computação* (influenciando um destes *broadcasts*) que foi iniciada em um processo depois de φ' e terminou em um outro processo mais

adiantado do que φ', e então φ' não seria um estado global.

No segundo caso, em φ' no mínimo um dos processos que participa do *breakpoint*, digamos $p_k \in N$, é tal que lp_k se tornou **true** pelo menos duas vezes. Neste caso, em cada um dos intervalos de tempo durante os quais lp_k = **false** precedendo φ', uma cadeia causal de mensagens de *computação* existe partindo de p_k e chegando em um outro processo que também participa do *breakpoint*, digamos $p_\ell \in N$, tal que lp_ℓ se torna **true** exatamente uma vez antes de φ'. Em q_ℓ, esta cadeia tem o efeito de provocar a atualização de $cp_\ell[k]$ com **false**, então o *broadcast*, iniciado por q_k quando lp_k se torna **true**, para permanecer **true** através de φ', provoca mudanças em cp_ℓ e é por isso propagado pela ação (3). Isto pode acontecer ou antes que lp_ℓ se torne **true** e então o *broadcast* iniciado por q_ℓ quando lp_ℓ se torna **true** possui $cp_\ell[k]$ = **true**, ou pode acontecer depois que lp_ℓ se torna **true**, neste caso as mensagens de $broadcast_k$, que q_ℓ encaminha, seguem as mensagens de $broascast_\ell$ enviadas anteriormente, possivelmente atualizando com **true** todo $cp_i[k]$ = **false** que possa ser encontrado para algum $p_i \in N$. O argumento a partir daí prossegeu como no primeiro caso.

(b) $\Rightarrow$ (c):

Pelos Lemas 1 e 2, os vetores alt_gs_i produzido pelas ações (1) e (2),o gs_i e alt_gs_i obtidos na ação (3), o $alt\_gs_i(lt_i)$ usado na ação (4), e o alt_gs_i produzido pela ação (5) são todos estados globais. Uma conseqüência disto é que, pela ação (3), no tempo t, gs_i é um estado global no qual $cp_i[k]$ = **true** para todo $p_k \in N$. Pode-se mostrar que gs_i é o estado global mais adiantado com estas características, considerando qualquer outro vetor de n componentes de tempos locais, chamado φ, tal que lp_k = **true** no tempo $\varphi[k]$

para todo $p_k \in N$ e além disso, $\varphi[k] < gs_i[k]$ para no mínimo um $p_k \in N$. Para este particular p_k, $gs_i[k]$ só pode ter recebido um valor maior do que $\varphi[k]$ em um dos dois casos.

No primeiro caso, uma cadeia causal de mensagens de *computação* deve ter existido de p_k para algum $p_\ell \in N$, que deve ter deixado p_k no tempo $gs_i[k]$ quando em q_k $alt\_cp_k[k]$ era **true** e deve ter chegado em q_ℓ quando $cp_\ell[\ell]$ = **false** mais adiantado do que $\varphi[\ell]$ e conseqüentemente φ não é um estado global em que lp_k = **true** para todo $p_k \in N$.

No segundo caso, a cadeia causal de mensagens de *computação* partindo de p_k foi iniciada quando $alt\_cp_k[k]$ = **false** em q_k, embora $cp_\ell[\ell]$ fosse como no primeiro caso, ou seja, $cp_\ell[\ell]$ = **false** mais adiantado do que $\varphi[\ell]$. Pela ação (5) e porque assumimos canais FIFO, $alt\_cp_\ell[k]$ deve ter se tornado **false** e permanecido assim até que um novo *broadcast* fosse iniciado por q_k no tempo $gs_i[k]$, quando $alt\_cp_k[k]$ se tornou **true**. Este *broadcast* pode ter atingido q_ℓ antes ou depois que lp_ℓ se tornasse **true**. No primeiro caso, o *broadcast* que q_ℓ então iniciou carregou $alt\_cp_\ell[k]$ = **true**, que ao atingir q_i e encontrar $cp_i[\ell]$ = **false**, pela ação (3), atualizou $cp_i[k]$ com **true**. No último caso, o *broadcast* que q_ℓ iniciou carregou $alt\_cp_\ell[k]$ com **false** e provocou a atualização de $cp_i[k]$ com **false**. Mas a mensagem encaminhada por q_ℓ, ao receber o *broadcast* originado por q_k, que atualizou $alt\_cp_\ell[k]$ com **true**, carregou este valor para q_i, e então $cp_i[k]$ foi atualizado com **true**. Mas, então, como no primeiro caso, φ não é um estado global em que lp_k = **true** para todo $p_k \in N$.

(c) $\Rightarrow$ (a):

Esta parte da prova é imediata.

■

Teorema 12. O algoritmo DETECT_CP possui complexidade de mensagens de $O((c + Pne)n\log T)$ *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.

Prova: Todo nó pode iniciar até P *broadcasts*, onde cada um deles é encaminhado por cada nó no máximo duas vezes, pela ação (3) e considerando que nenhuma mensagem de um dado *broadcast* chegando em q_i atualiza gs_i ou alt_gs_i (da mesma forma, cp_i ou alt_cp_i) mais de uma vez. A complexidade de mensagens é de $O((c + Pne)n\log T)$ *bits*, obtido da mesma forma que a complexidade do Teorema 4, considerando ainda o fato de que no pior caso cada *broadcast* passa por cada canal duas vezes em cada direção. As outras medidas são dadas diretamente pelo Teorema 10.

■

Capítulo 8

Problemas Relacionados

Embora o projeto de algoritmos distribuídos para detecção de predicados globais seja o objetivo principal deste trabalho, neste capítulo são apresentados ainda dois outros aspectos relacionados com este problema.

O primeiro aspecto se refere à possibilidade de parar a computação no estado global detectado pelos algoritmos apresentados. Para isto, sugerimos as técnicas de *checkpointing* e *rollback-recovery* e discutimos estratégias adicionais para economizar memória enquanto os *checkpoints* são gravados.

O segundo aspecto se refere à avaliação de expressões nos estados globais detectados, ou melhor, todos os predicados globais discutidos anteriormente podem ser utilizados para definir estados globais onde se deseje avaliar uma determinada expressão, que pode ser constituída de variáveis de todos os processos que participam da computação. Para isto, as variáveis devem ser salvas diversas vezes no decorrer da computação quando mudam de valor.

Como as estratégias utilizadas no primeiro e segundo casos para economizar memória nas gravações de *checkpoints* e variáveis, respectivamente, são

semelhantes, trataremos os dois assuntos neste mesmo capítulo.

8.1 Checkpointing e Rollback Recovery

As técnicas de *checkpointing* e *rollback recovery* permitem que os processos continuem as suas execuções apropriadamente depois de falhas, sem a necessidade de recomeçarem do início [20]. Embora originalmente formuladas no contexto de tratamento de falhas, estas técnicas podem também ser usadas na depuração de programas distribuídos, particularmente em associação com os algoritmos de detecção de *breakpoint* discutidos neste trabalho. Todos os algoritmos detectam estados globais com certas propriedades de interesse, mas depois da detecção de tal estado global, pelo processo, digamos, $q_i \in N$, a informação desta detecção deve ser difundida através dos outros processos a fim de que eles possam parar (já que este é o objetivo de se estabelecer *breakpoints*). Entretanto, esta difusão da ordem de parada necessariamente atinge a maior parte dos nós quando eles estão em estados locais mais tardios do que aqueles gravados no estado global detectado. Neste caso, então, pode-se empregar as técnicas de *checkpointing* e *rollback-recovery*, porque a maior parte dos processos, ao receber a ordem de parada, será forçada a retornar para um ponto anterior de sua execução, a fim de que o sistema possa ser parado no estado global desejado.

A gravação de *checkpoints* enquanto os processos executam constitui uma área de pesquisa com seus próprios problemas. Existem basicamente duas abordagens para esta questão. Na primeira, os *checkpoints* são gravados independentemente em cada nó e quando existe a necessidade de se retornar

o sistema para um estado anterior o conjunto de *checkpoints* do sistema é examinado a fim de se obter aqueles que formam um estado global (por exemplo, [5]). A segunda solução é mais conservadora, e tenta assegurar que todo *checkpoint* gravado localmente seja parte de um *checkpoint* do sistema que constitui um estado global. Neste caso, voltar o sistema é um problema simples, já que os *checkpoints* do sistema disponíveis são estados globais (por exemplo, [19], [34]).

A técnica proposta para usar junto com os algoritmos de detecção de *breakpoint* é como a do primeiro tipo que foi descrito acima, embora felizmente sem o risco de que os *checkpoints* do sistema não constituam um estado global. A técnica geral é bastante simples e se aproveita do fato de que todo componente no gs_i que q_i detecta para algum $p_i \in N$ seja ou igual a zero, ou o tempo local no qual um processo teve a condição local com a qual participa no *breakpoint* satisfeita, ou ainda o tempo local no qual uma mensagem de *computação* foi enviada. Desta forma, se todo processo grava um *checkpoint* no começo de sua computação, um outro quando sua condição local é satisfeita e um outro sempre que enviar uma mensagem de *computação*, então uma vez que gs_i é detectado por q_i todo q_k tal que $p_k \in N$ tem simplesmente que retornar p_k de volta para o tempo local $gs_i[k]$.

Certamente, esta estratégia aumenta as necessidades de memória em relação às dadas pelos Teoremas 4, 8, 10 e 12, mas algumas melhorias podem ser empregadas à estratégia geral, a fim de minimizar as necessidades de memória em cada processo. Por exemplo, para a detecção de um *breakpoint* sobre um predicado disjuntivo, q_i grava *checkpoints* somente até lp_i se tornar `true` (se p_i estiver realmente participando do *breakpoint*) ou até que

uma mensagem de *computação* carregando bit_j = **true** seja recebida de q_j tal que $p_j \in N_i$ (sem considerar se p_i participa no *breakpoint* ou não). Em qualquer um dos casos, somente o *checkpoint* cujo tempo de gravação seja o $gs_i[i]$ após atualização precisa ser armazenado, enquanto os outros podem ser descartados.

Para a detecção de um *breakpoint* incondicional, um processo $p_i \in N$ que participa no *breakpoint* precisa de um único *checkpoint* no tempo lub_i. Se p_i não participa no *breakpoint*, então q_i, ao receber uma mensagem de *broadcast* carregando gs_j de q_j tal que $p_j \in N_i$ não precisa mais gravar *checkpoint* ao enviar mensagens de *computação* para q_j se $ub_j[j]$ = **true** e pode, além disso, descartar todo *checkpoint* gravado antes do que $gs_j[i]$.

Para a detecção de *breakpoints* sobre predicados conjuntivos, um processo q_i tal que $p_i \in N$ grava *checkpoints* quando lp_i se torna **true** e ao enviar mensagens de *computação*, mas somente enquanto lp_i = **true** (se p_i participa no *breakpoint*) ou então sempre (se p_i não participa no *breakpoint*, sendo por isso considerado como se seu predicado fosse sempre **true**). Se o predicado conjuntivo é estável, o recebimento de uma mensagem de *broadcast* carregando gs_j de q_j tal que $p_j \in N_i$ permite que q_i descarte todos os *checkpoints* gravados antes de $gs_j[i]$. Este caso é similar ao caso de *breakpoints* incondicionais, com a diferença de que q_i deve continuar com a gravação de *checkpoints* ao enviar mensagens de *computação* para q_j porque, em contraste com o primeiro caso, o tempo local com o qual p_j participa no estado global desejado pode ainda não ter ocorrido (mesmo que este já tivesse atingido uma situação em que seu predicado local fosse verdadeiro). Se o predicado conjuntivo não é estável, então tudo que pode ser feito é descartar todo *checkpoint* gravado

antes de $gs_i[i]$, após atualização, já que definitivamente eles não farão parte no estado global em que o predicado é satisfeito.

A estratégia que foi apresentada até agora precisa ser melhorada para que o *checkpoint* global inclua mensagens em trânsito também. Esta medida, embora não seja essencial para o processo de *rollback*, é essencial para reiniciar a execução depois que o sistema tiver sido examinado no *breakpoint*. Algumas soluções têm sido discutidas, como por exemplo em [33] e [34].

8.2 Avaliação de Expressões

Além de *breakpoints*, outra facilidade normalmente oferecida pelos depuradores seqüenciais é a de avaliação de expressões, constituídas de variáveis do programa e operadores aritméticos ou lógicos, em determinados pontos do programa. A estes pontos, da mesma forma que no caso de *breakpoints*, podemos associar condições. Em programas distribuídos, todos os predicados discutidos previamente podem ser utilizados como condições sob as quais o depurador deverá calcular a expressão que pode ser constituída de variáveis de qualquer processo que participa da computação. Assim, quando ocorre a detecção de um estado global, pelo processo, digamos, $q_i \in N$, a informação desta detecção deve ser difundida através dos outros processos a fim de que eles possam fornecer os valores de suas variáveis ao processo q_i para que então ele calcule a expressão (considerando que todos os processos $q_i \in N$ conheçam a expressão). Entretanto, esta difusão pode atingir os nós quando eles estão em estados locais mais adiantados do que aqueles gravados no estado global detectado e, por isso mesmo, não há garantia de que os valores das variáveis

sejam os mesmos. Como resultado disso, a solução empregada consiste em periódicas gravações das variáveis com os respectivos tempos locais a fim de que se possa recuperá-las quando o predicado global for satisfeito.

A técnica geral é idêntica à vista anteriormente e assim, também, toma vantagem do fato de que todo componente no gs_i que q_i detecta para algum $p_i \in N$ seja ou igual a zero, ou o tempo local no qual um processo teve a condição local com a qual participa no *breakpoint* satisfeita, ou ainda o tempo local no qual uma mensagem de *computação* foi enviada. Portanto, se todo processo grava as suas variáveis envolvidas na expressão no começo de sua computação, quando sua condição local é satisfeita, e sempre que envia uma mensagem de *computação*, então uma vez que gs_i é detectado por q_i todo q_k tal que $p_k \in N$ é capaz de recuperar as suas variáveis que foram gravadas com o tempo local $gs_i[k]$. As melhorias que podemos empregar à estratégia geral são basicamente as mesmas já descritas na seção anterior, com apenas mais um detalhe, que se refere ao fato de que as variáveis locais podem ser iguais entre duas gravações consecutivas. Assim, podemos evitar uma gravação sempre que o valor da variável a ser gravada for igual ao da última gravação. A implicação desta melhoria adicional é que na recuperação pode não haver variável gravada para o tempo local fornecido por $gs_i[k]$. Neste caso, a estratégia é recuperar a variável gravada no maior tempo que preceder o tempo local dado por $gs_i[k]$.

Capítulo 9

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. A facilidade de se estabelecerem *breakpoints* onde o contexto do programa possa ser analisado representa uma das características mais importantes de depuradores de programas paralelos distribuídos.

Apresentamos e analisamos, em relação à corretude e à complexidade, quatro algoritmos de detecção de *breakpoints*, especificamente um para detecção de *breakpoints* incondicionais (algoritmo DETECT_UBP) e três para detecção de *breakpoints* condicionais. No caso de *breakpoints* condicionais, um deles é utilizado para a detecção de *breakpoints* sobre predicados disjuntivos (algoritmo DETECT_DP), o outro para a detecção de predicados conjuntivos estáveis (algoritmo DETECT_STABLE_CP) e finalmente um para a detecção de predicados conjuntivos genéricos (algoritmo DETECT_CP).

Em relação às contribuições deste trabalho, podemos citar como a principal o fato de os algoritmos projetados serem os primeiros totalmente dis-

tribuídos 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.

Embora no Capítulo 4 tenhamos apresentado uma breve discussão de outros algoritmos relacionados, evitamos qualquer comparação baseada na complexidade, já que os algoritmos apresentados são inerentemente diferentes. Exceto pelo algoritmo DETECT_DP, cuja complexidade de tempo global é de $O(1)$, todos os algoritmos possuem complexidade de tempo global $O(n)$. Todos os quatro algoritmos possuem a mesma complexidade de tempo local de $O(n)$ por mensagem recebida. A necessidade de memória de todos os algoritmos é de $O(n\log T)$ *bits* por processo,onde T é um limite superior do tempo local de qualquer processo. Se a computação propriamente emprega $O(c)$ mensagens, então o algoritmo DETECT_DP possui uma complexidade de mensagens de $O(cn\log T)$ *bits*, enquanto que os algoritmos DETECT_UBP e DETECT_STABLE_CP possuem a mesma complexidade de mensagens de $O((c+ne)n\log T)$ *bits*. O algoritmo DETECT_CP possui a complexidade de mensagens de $O((c+Pne)n\log T)$ *bits*.

Tendo em vista parar a computação no estado global detectado, sugerimos 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.

Neste trabalho ainda foi apresentado um resumo das principais técnicas de depuração de programas paralelos visando permitir ao leitor uma melhor compreensão do contexto no qual o problema que foi tratado nesta tese está inserido.

Como sugestões para futuros trabalhos, poderíamos projetar algoritmos distribuídos que detectassem outros tipos de predicados. Uma importante contribuição seria o projeto de algoritmos que detectassem outros tipos de predicados relacionais, além do tipo resolvido por Tomlinson e Garg [43], onde são tratados apenas os predicados da forma $(x_0 + x_1 < C)$, onde x_0 e x_1 são valores inteiros nos processos P_0 e P_1 em um sistema de N processos, ou seja, o predicado é restrito a apenas duas variáveis. Outro problema que também poderia ser tratado é aquele considerado por Cooper e Marzullo [8], que tenta detectar se um predicado é satisfeito para alguma execução do programa, ou se é satisfeito para toda execução do programa. Na verdade os

algoritmos tratados nesta tese e em todos os outros trabalhos relacionados fazem a detecção para apenas um perfil da execução e Cooper e Marzullo não conseguem soluções eficientes para este problema.

Bibliografia

[1] Allen, T. R., and Padua, D. A. Debugging FORTRAN on a shared memory machine. *Proc. of the International Conference on Parallel Processing*, 1987, pp. 721–727.

[2] Allen, R., Baumgartner, D., Kennedy, K., and Porterfield, A. Ptoll: A semiautomatic parallel programming assistant. *Proc. of the International Conference on Parallel Processing*, 1986, pp. 164–170.

[3] Banerjee, U., Chen, S., Kuck, D. J., and Towle, R. A. 1979. Time and parallel processors bounds for fortran-like loops. *IEEE Trans. Comput. 28* (1979), 660–670.

[4] Becher, J. D., and McDowell, C. E. Debugging the MP-2001. *Proc. of New Frontiers, A Workshop on Future Directions of Massively Parallel Processing*, 1992, pp. 48–57.

[5] Bhargava, B., and Lian, S. R. Independent checkpointing and concurrent rollback for recovery — An optimistic approach. *Proc. of the IEEE Symposium on Reliable Distributed System*, 1988, pp. 3–12.

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

[7] Choi, J., Miller, B. P., and Netzer, R. H. B. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. on Programming Languages and Systems 13* (1991), 491–530.

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

[9] Fidge, C. Logical time in distributed computing systems. *IEEE Computer 24* (1991), 28–33.

[10] Garcia-Molina, H., Germano, F., and Kohler, W. Debugging a distributed computing system. *IEEE Trans. on Software Engineering 10* (1984), 210–219.

[11] 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.

[12] 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.

[13] 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.

[14] Harter, P. K., Heimbigner, D. M., and King, R. IDD: An interactive distributed debugger. *Proc. of the fifth International Conference on Distributed Computing Systems*, 1985, pp. 498–506.

[15] Helmbold, D., and McDowell, C. Determining possible event orders by analyzing sequential traces. *IEEE Trans. on Parallel and Distributed Systems 4* (1993), 827–840.

[16] Hough, A. A., and Cuny, J. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. *Proc. of the International Conference on Parallel Processing*, 1987, pp. 735–738.

[17] Hurfin, M., Plouzeau, N., and Raynal, M. Detecting atomic sequences of predicates in distributed computations. *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993, pp. 32–40.

[18] Joyce, J., Lomow, G., Slind, K., and Unger, B. W. Monitoring distributed systems. *ACM Trans. on Computer Systems 5* (1987), 121–150.

[19] Kim, K. H., You, J. H., and Abouelnaga, A. A scheme for coordinated execution of independently designed recoverable distributed processes. *Proc. of the IEEE Fault-Tolerant Computing Symposium*, 1986, pp. 130–135.

[20] Koo, R., and Toueg, S. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software engineering 13* (1987), 23–31.

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

[22] LeBlanc, T., and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Trans. on Computers 36* (1987), 471–482.

[23] LeBlanc, T., and Mellor-Crummey, J. M., Fowler, R. Analysing parallel program executions using multiple views. *J. of Parallel and Distributed Computing 9* (1990), 203–217.

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

[25] Manning, C. R. Traveler: The apiary observatory. *Proc. of European Conference on Object Oriented Programming*, 1987, pp. 97–105.

[26] Mattern, F. Virtual time and global states of distributed systems. *Proc. of the 1988 International Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[27] McDowell, C. Viewing anomalous states in parallel programs. *Proc. of the International Conference on Parallel Processing*, 1988, pp. 54–57.

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

[29] 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.

[30] Miller, B., and Choi, J. A mechanism for efficient debugging of parallel programs. *Proc. of the Sigplan'88 Conference on Programming Language Design and Implementation*, 1988, pp. 135–144.

[31] Netzer, R. H. B. Optimal tracing and replay for debugging shared-memory parallel programs. *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging* (1993).

[32] Netzer, R. H. B., and Miller, B. Optimal tracing and replay for debugging message-passing parallel programs. *Proc. of Supercomputing'92* (1992).

[33] Netzer, R. H. B., and Xu, J. Adaptive message logging for incremental program replay. *IEEE Parallel & Distributed Technology 1* (1993), 32–39.

[34] Ramanathan, P., and Shin, K. G. Use of common time base for checkpointing and rollback recovery in a distributed system. *IEEE Trans. on Software Engineering 19* (1993), 571–583.

[35] Scheifler, R. W., and Gettys, J. The X window system. *ACM Trans. Graph. 5* (1986), 2.

[36] Seidner, R., and Tindall, N. Interactive debug requirements. *SIGPLAN Notices 9* (1983), 9–22.

[37] Socha, D., Bailey, M. L., and Notkin, D. Voyeur: Graphical views of parallel programs. *Proc. of Workshop on Parallel and Distributed Debugging.* Publicado na *SIGPLAN Notices 24* (1989), 206–215

[38] 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.

[39] 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.

[40] Stone, J. M. A graphical representation of concurrent processors. *Proc. of Workshop on Parallel and Distributed Debugging.* Publicado na *SIGPLAN Notices 24* (1989), 226–235.

[41] Sun microsystems. *NeWS Preliminary Technical Overview*, 1986.

[42] Taylor, R. N., and Osterweil, L. J. Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. on Software Engineering 6* (1980), 265–278.

[43] 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.

[44] Xu, J., and Netzer, R. H. B. Adaptive independent checkpointing for reducing rollback propagation. *Proc. of the Fifth IEEE Symposium on Parallel and Distributed Processing*, 1993, pp. 754–761.