Depuração de Programas Paralelos

Benhur de Oliveira Stein[1]

Universidade Federal de Santa Maria
Centro de Tecnologia
Departamento de Eletrônica e Computação
97115-900 Santa Maria
E-mail: benhur@inf.ufsm.br

1 Introdução

A depuração de programas paralelos tem sido uma tarefa difícil há bastante tempo. Nas primeiras máquinas paralelas com memória distribuída havia uma ausência quase total de sistema operacional e de ferramentas de *software*. Essa situação melhorou bastante, visto que atualmente cada nó de uma máquina paralela ou de uma rede de estações de trabalho oferece o conjunto de funcionalidades de um sistema operacional completo. No entanto, ainda existem os problemas específicos ao paralelismo.

Além da depuração de programas para a correção de erros lógicos, a programação paralela exige ainda a depuração visando melhorar o desempenho. A depuração do desempenho é uma fase importante no ciclo de desenvolvimento de programas paralelos, uma vez que o principal motivo para se utilizar sistemas paralelos é a obtenção de alto desempenho. A depuração do desempenho geralmente inclui várias fases: monitoramento para recolher dados de desempenho, análise de dados e apresentação dos índices de desempenho ao programador, geralmente por ferramentas de visualização sofisticadas.

Nosso interesse inicial está nas ferramentas de base, que são os depuradores simbólicos, depois nos trabalhos consagrados aos problemas principais causados pelo paralelismo: o não determinismo aparente das execuções paralelas, a dificuldade de observação de um grande número de processos ou de um estado global coerente. Na seção 3, são apresentadas técnicas de monitoramento de programas paralelos, de análise e correção dos dados recolhidos, e finalmente de apresentação desses dados ao programador.

2 Ferramentas de depuração

2.1 Depuradores simbólicos paralelos

Os depuradores paralelos [26, 1] generalizam as funções clássicas dos depuradores seqüenciais para vários processos. A principal diferença em relação aos depuradores seqüenciais consiste na noção de *contexto*: as funcionalidades herdadas dos depuradores sequenciais afetam somente os processos que fazem parte do contexto corrente.

[1] Este texto é "fortemente baseado" nas notas de aula de Jacques Chassin de Kergommeaux, intituladas "Parallélisme : compilation et environnement d'exécution".

Os depuradores paralelos permitem que os programadores controlem a execução de um programa paralelo: colocação de pontos de parada em um processo ou em um conjunto de processos, execução passo a passo de um processo, etc. Oferecem também a possibilidade de se observar o estado da pilha de cada processo, bem como os valores de suas variáveis, possivelmente através da ajuda de um sistema de janelas.

Ferramentas desse tipo facilitam bastante a depuração de programas paralelos e, de fato, figuram no catálogo da maioria dos fabricantes de máquinas paralelas. Na falta de uma tal ferramenta, às vezes é possível simular seu funcionamento conectando-se um depurador seqüencial a cada processo da aplicação.

Nem todos os depuradores paralelos existentes sabem gerenciar programas que se utilizam da multiprogramação leve nos nós do sistema paralelo. No pior dos casos, eles são incompatíveis com a existência de múltiplos fluxos de execução em cada nó, tornando necessária a utilização de técnicas mais "rústicas" de depuração. Em alguns casos, o depurador é compatível com a utilização de múltiplos fluxos, mas não necessariamente suporta a sua depuração. Seguir a execução em um nó neste caso pode se tornar mais complexo, devido às trocas de contexto entre fluxos de execução, que introduzem rupturas no controle não visíveis no código fonte do programa.

Para se depurar confortavelmente aplicações paralelas que utilizam multiprogramação leve em seus nós, é necessário que o depurador simbólico forneça um suporte explícito à noção de fluxos de execução. O depurador *xpdbx* das máquinas IBM SP [26] possui essa característica, que permite ao programador seguir explicitamente a passagem de um fluxo a outro no interior do nó corrente (ver figura 1).

Diante da grande variedade de interfaces e funcionalidades disponíveis nos vários depuradores simbólicos paralelos existentes (como, por exemplo, Cdbx [9], codeview [46], HP/DDE [8], P2D2 [25], TotalView [1] entre outros), instalou-se um fórum para o desenvolvimento de padrões para depuradores simbólicos de programas paralelos, resultando no *High Performance Debugging Standard* [13], que deve ser adotado nos próximos depuradores desenvolvidos. A existência de um padrão deve diminuir as dificuldades de adaptação a um novo ambiente de depuração, tarefa freqüente na comunidade de programação paralela, devido à constante troca de plataforma de desenvolvimento, na busca incessante pelo melhor desempenho.

Este padrão define uma funcionalidade mínima e uma sintaxe para se explorar essa funcionalidade. Cobre também os casos de depuração de programas contendo vários fluxos de execução em um processador, programas utilizando múltiplos processos com um fluxo de execução por processo e também o caso misto, programas compostos por vários processos possivelmente com múltiplos fluxos de execução. Além disso, ele define as possíveis semânticas de controle da execução de fluxos, como por exemplo paradas de fluxos de execução em *actionpoints* (*breakpoints*, *watchpoints* ou barreiras), ou a continuação da execução de fluxos parados pelo depurados. Os estados possíveis para os fluxos e as ações executáveis sobre os fluxos em cada estado, e funcionalidades para limitar o volume de dados apresentados pelo depurador ao programador (por exemplo,

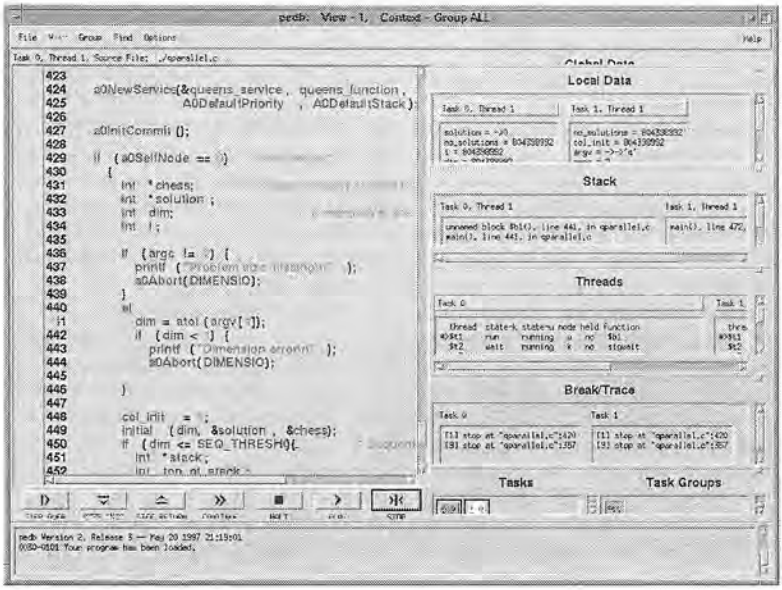

Figura 1. Exemplo de utilização do depurador simbólico xpdbx do IBM SP

se o programador pede para ver o valor de um vetor que existe em vários processos, o depurador só apresentará o valor naqueles processos em que eles são diferentes, não apresentando várias vezes o mesmo valor, se vários processos têm o vetor idêntico) são também definidas pelo padrão.

Desenvolvidos para a depuração "fina" de programas, os depuradores apresentam dificuldade em apresentar ao programador dados mais globais da execução de seus programas, e também da evolução da execução do programa no tempo. Para estas tarefas, uma ferramenta de visualização é muito mais adequada. Além disso, qualquer que seja o seu grau de sofisticação, os depuradores simbólicos paralelos são insuficientes para lutar contra erros intermitentes que aparecem em programas não determinísticos, como apresentado a seguir.

2.2 Problema do não determinismo

Um grande número de programas paralelos apresentam um comportamento não determinístico. Pode-se distinguir dois tipos de não determinismo: interno e externo. Um pro-

grama apresenta determinismo **externo** se ele produz sempre as mesmas saídas quando apresentado às mesmas entradas. Quando, além de produzir as mesmas saídas, a seqüência de instruções executada por cada processo e os valores dos operandos de cada instrução são também determinísticos, diz-se que o programa apresenta determinismo **interno**.

Um comportamento não determinístico provém essencialmente da existência de **condições de concorrência** (*race conditions*). A **concorrência de sincronização** deve-se à competição entre os processos para entrar em uma seção crítica ou obter um semáforo ou ainda quando duas mensagens, simultaneamente em trânsito na rede de comunicação, podem ser recebidas por um mesmo comando de recepção [39] (veja figura 2). Uma **concorrência por dados** é produzida quando vários processos realizam acessos simultâneos aos mesmos dados sem utilizar mecanismos adequados de controle. As concorrências por dados são consideradas erros de programação e podem ser eliminadas com a adição de sincronizações adequadas entre os processos concorrentes. Por outro lado, eliminar completamente as concorrências de sincronização para tornar os programas inteiramente determinísticos teria o efeito de interditar várias técnicas de programação perfeitamente válidas para adaptação do programa a seu ambiente de execução. Um exemplo clássico é um processo servidor que espera requisições de qualquer cliente.

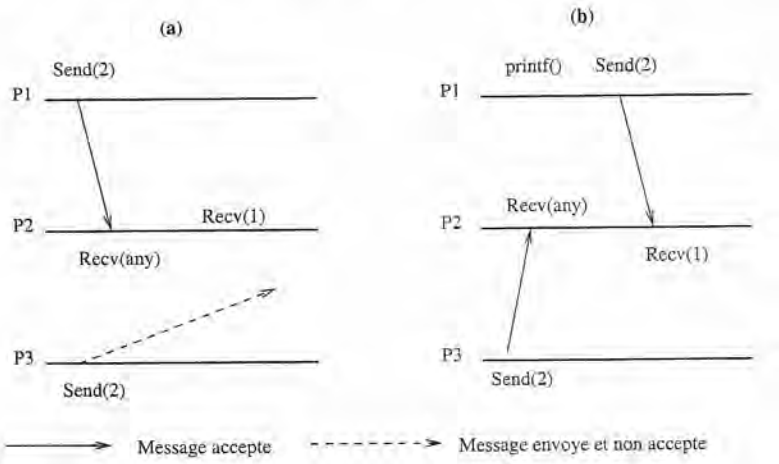

Figura 2. Competição em um programa paralelo (de [10])

Vários trabalhos analisaram o problema de detectar as condições de concorrência em programas paralelos [22]. A dificuldade de detecção depende do modelo de programação

utilizado. Em vários casos, a detecção estática (antes da execução do programa) não tem solução. Soluções aproximadas para esse problema devem evitar, por um lado, detectar um número exagerado de condições de concorrência (em sua maioria válidas), e por outro lado, não detectar as condições de concorrência realmente erradas. Algumas técnicas de detecção dinâmica foram desenvolvidas [45]. Em alguns casos, tais técnicas permitem identificar os erros que aparecem durante uma execução particular de um programa paralelo.

Um programa com condições de concorrência pode ser bastante sensível ao ambiente de execução e portanto a um grande número de fatores sobre os quais o programador não tem controle: o conteúdo inicial das memórias *cache*, o estado inicial do sistema operacional ou ainda a execução simultânea de outros processos no sistema, por exemplo. O comportamento indeterminístico de um programa incorreto pode ser a origem de erros intermitentes, que aparecem com uma freqüência baixa ou desaparecem quando se insere instruções para a geração de rastros ou se utilize um depurador simbólico (em razão das mudanças no comportamento do programa paralelo induzidas pela utilização de tais ferramentas). Esse tipo de comportamento impossibilita a depuração "cíclica", utilizada normalmente em programas seqüenciais, onde um programa incorreto pode ser reexecutado tantas vezes quantas forem necessárias para localizar um erro, colocando-se instruções de geração de rastros ou pontos de parada cada vez mais próximos do erro observado.

Classicamente, a técnica mais utilizada para detectar os erros intermitentes consiste em **registrar** uma execução inicial e **forçar** as execuções seguintes a se comportarem de uma forma determinística em relação à execução inicial, utilizando as informações registradas. Desta forma, a depuração de um programa incorreto pode utilizar as técnicas de depuração cíclicas, reexecutando-se sucessivamente o programa de acordo com o registro de uma execução onde o erro tenha se manifestado. Uma condição necessária ao uso desta técnica é que a perturbação causada aos processos pelo registro inicial seja tão pequena quanto possível para que os erros que aparecem em uma execução "normal" não desapareçam quando o programa for executado em "modo de gravação", impossibilitando sua reprodução. Se a sobrecarga causada pelo registro dos rastros de uma execução for suficientemente pequena, é possível deixar ativada a opção de registrar em todas as execuções do programa paralelo, de forma que todo erro, mesmo que seja pouco freqüente, possa ser registrado e reproduzido à vontade.

2.3 Reexecução determinística de programas paralelos

O princípio da reexecução determinística é baseado no comportamento determinístico de cada processo visto individualmente, quando ele recebe os mesmos dados de entrada, na mesma ordem. Duas técnicas foram desenvolvidas para permitir a reexecução determinística de programas paralelos: a primeira é baseada nos **dados**, gravando o conteúdo de todas as entradas dos processos que compõem a aplicação paralela; a segunda, mais eficiente, é baseada no **controle**: é o mecanismo *Instant Replay* de Leblanc e Mellor-

Crummey [11, 29]. Essa segunda técnica baseia-se na observação de que é suficiente registrar a *ordem* dos acessos a objetos compartilhados a fim de se poder reproduzir execuções "equivalentes". Intuitivamente, duas execuções X e Y de um programa P serão consideradas equivalentes se cada processo de P passa pela mesma seqüência de estados nas duas execuções. Duas execuções equivalentes produzirão o mesmo resultado, ou exibirão o mesmo erro [31]. O fato de serem equivalentes não implica necessariamente que sejam idênticas: nos modelos de execução determinísticas existentes a ordem relativa das execuções das instruções elementares pelos processadores não tem nenhuma importância. O restante desta seção trata somente das técnicas de reexecução determinística baseadas no controle.

Princípio de *Instant Replay* O mecanismo do *Instant Replay* é baseado na hipótese do comportamento determinístico de cada processo individualmente:

- se um processo recebe as *mesmas* entradas na *mesma* ordem,
- esse processo produzirá as *mesmas* saídas na *mesma* ordem,
- ... que serão utilizadas como entradas de outros processos.

Mais formalmente, Éric Leu demonstra em sua tese um teorema que determina uma condição suficiente para que duas execuções de um programa paralelo sejam equivalentes [31]. Esse teorema tem por hipótese que duas execuções partem de um mesmo estado inicial, e que todos os eventos das duas execuções são eventos elementares (cada evento depende somente do estado do processo onde o evento ocorre, anterior à sua ocorrência). Sob essas hipóteses, duas execuções X e Y de um programa paralelo serão equivalentes se **a ordem de recepção das mensagens por cada processo e a ordem parcial de acesso às unidades de memória compartilhada são idênticas nas duas execuções X e Y.**

Os mecanismos de *Instant Replay* asseguram a equivalência entre execuções forçando, nas reexecuções dos programas paralelos, cada processo a receber as mesmas mensagens na mesma ordem que foram recebidas durante a execução registrada inicialmente e/ou que os acessos às variáveis de sincronização sejam realizados na mesma ordem. Como as saídas dos processos são calculadas a cada reexecução, não é necessário registrar o conteúdo das mensagens na execução inicial, o que permite que a sobrecarga em tempo e em espaço da execução inicial seja limitada. Pode-se notar que a reexecução determinística só poderá garantir a equivalência entre execuções para programas cujas únicas condições de concorrência sejam as concorrências de sincronização, intencionais. Em geral não é possível forçar-se uma reexecução determinística na presença de condições de concorrência sobre os dados (erros de programação). O mecanismo de reexecução determinística foi no entanto utilizado para a implementação de um mecanismo de detecção de tais erros [45].

Exemplo de implementação Neste exemplo mostra-se uma implementação de reexecução determinística na qual as interações entre os processos são modeladas como acessos

a objetos compartilhados segundo um protocolo do tipo leitores-escritor [29]. Esse tipo de modelo pode igualmente ser adaptado aos casos onde os processos se comunicam por troca de mensagens.

Durante a execução inicial (fase RECORD), registra-se unicamente a *ordem* de acesso aos objetos compartilhados:

- à cada objeto é associado um número de versão,
- durante a fase inicial, cada objeto registra o número de versão de cada objeto que acessa e
- durante as reexecuções (fase REPLAY), força-se o processo à acessar as *mesmas* versões dos objetos.

Os valores trocados entre os processos dependem somente:

- dos valores iniciais dos objetos compartilhados,
- da ordem de acesso aos objetos pelos processos e
- do comportamento determinístico dos processos.

Além do número de versão associado a cada objeto compartilhado, associa-se a cada processo uma estrutura de dados linear, comparável a uma fita magnética, na qual se registra os números de versão dos objetos acessados, na ordem em que esses acessos ocorrem. Não é necessário registrar-se a identificação do objeto acessado. Para explicar o funcionamento do *Instant Replay*, é mais fácil fornecer os procedimentos de acesso aos dados compartilhados, em leitura e em escrita:

```
ReaderEntry (object, process)
  if mode = RECORD then
    lock(object.lock);
    atomic_add(object.activeReaders, 1);
    unlock(object.lock);
    write_tape(process, object.version);
  else /* mode = REPLAY */
    key = read_tape(process);
    wait until object.version = key;
  end if;
end ReaderEntry;

ReaderExit (object)
  atomic_add(object.totalReaders, 1);
  atomic_add(object.activeReaders, -1);
end ReaderExit;
```

```
WriterEntry (object, process)
  if mode = RECORD then
    lock(object.lock);
    wait until object.activeReaders = 0;
    write_tape(process, object.version);
    write_tape(process, object.totalReaders);
  else /* mode = REPLAY */
    key = read_tape(process);
    wait until object.version = key;
    key = read_tape(process);
    wait until object.totalReaders = key;
  end if;
end WriterEntry;

WriterExit (object)
  object.totalReaders = 0;
  if mode = RECORD then
    object.version++;
    unlock(object.lock);
  else /* mode = REPLAY */
    atomic_add(object.version, 1);
  end if;
end WriterExit;
```

Além do registro dos valores dos contadores associados aos objetos compartilhados, é necessário registrar-se as escolhas não determinísticas feitas pelo programa, assim como os resultados da utilização de funções que dependem do contexto (leitura do valor do relógio, por exemplo).

2.4 Complexidade das execuções paralelas

Diante de um grande número de processos paralelos, as ferramentas de depuração não oferecem auxílio suficiente à compreensão global de uma aplicação paralela. Uma forma interessante de atacar esse problema foi proposta por Leu e Schiper [30], e consiste em conjugar a reexecução determinística à ferramenta de visualização ParaGraph, utilizada normalmente para a medida de desempenho: ParaGraph oferece uma visualização de alto nível da execução do programa paralelo (processos ativos, comunicação entre processos, etc.), enquanto o depurador paralelo permite o exame de "baixo nível" da aplicação. O ambiente Annai [7, 5] integra depuração e medidas de desempenho mostrando-se como um dos ambientes de desenvolvimento de programas paralelos mais completos existentes atualmente.

As dificuldades impostas pela geração de rastros de aplicações paralelas irregulares tornam esse tipo de conjugação mais difícil de se realizar caso os rastros não sejam ordenados (ver seção 3.4) e caso as datas dos eventos devam ser corrigidas *post mortem* [35].

2.5 Obtenção de estado global

A possibilidade de se obter um estado global de uma execução paralela (*snapshot*) pode se revelar particularmente útil para a depuração. Ela permite, em particular, testar-se uma propriedade global sobre o conjunto dos processadores que participam da execução paralela considerada. Ela pode também servir de base à implementação de um mecanismo de recuperação de um estado salvo do programa, o que pode ser muito útil em um ciclo de depuração. O principal problema para se obter um estado global de uma execução paralela ou distribuída é a coerência entre os estados dos vários processadores observados. Devido aos atrasos na propagação da informação, a obtenção de um estado global não é instantânea e os estados salvos poder resultar incoerentes. Uma incoerência possível é, por exemplo, a existência de uma mensagem recebida no estado gerado por um processo receptor enquanto que o estado do processo emissor correspondente foi gerado antes da emissão dessa mensagem.

Vários algoritmos foram propostos na literatura para garantir a geração de estados globais coerentes em sistemas paralelos e distribuídos (veja [21, 37]). Pode-se distinguir os algoritmos coordenados, nos quais um processo particular inicia e controla a geração do estado global, e os algoritmos não coordenados, onde se detecta um estado global coerente a partir de estados salvos localmente de forma independente. Um exemplo de um algoritmo coordenado [37]:

1. um processo é inicialmente verde, assim como as mensagens que ele envia.
2. após salvar seu estado, um processo torna-se vermelho, e as mensagens que ele envia também.
3. quando recebe uma mensagem vermelha, um processo verde deve salvar seu estado.
4. cada processo conta o valor da diferença entre o número de mensagens verdes enviadas e recebidas.
5. cada processo vermelho envia seu estado local e o valor da diferença a um processo particular chamado coletor.
6. cada processo vermelho envia ao coletor as mensagens verdes recebidas.

2.6 Como se depura um programa paralelo

As ferramentas de base para a depuração de programas paralelos são extensões aos depuradores sequenciais tradicionais, que permitem controlar a execução de cada processo da aplicação. As técnicas de execução determinística, combinadas com as ferramentas clássicas durante as fases de reexecução, apresentam um auxílio considerável na busca de erros intermitentes. As ferramentas de depuração têm em comum com as ferramentas de medida de desempenho a necessidade de representar de uma forma inteligível e coerente uma grande quantidade de informações (veja seção 3.7). Torna-se então desejável a integração do conjunto de ferramentas de auxílio à depuração em um ambiente comum que ofereça uma interface única aos programadores. Um bom exemplo de ambiente desse tipo é Annai [5] (que no entanto não trata o problema da intrusão).

Apesar da multiplicidade das ferramentas e ambientes de auxílio à depuração de programas, seu uso ainda é restrito. O "método" mais clássico para depurar um programa paralelo ainda consiste em inserir no programa comandos de impressão (`printf`), enquanto o desempenho dos programas paralelos é geralmente avaliado pela inserção de instruções *ad hoc* por seus autores [41]. Os motivos deste fenômeno são múltiplos mas é em parte explicado pelo grande número de modelos de programação paralela existentes e pela falta de profissionalismo de certas ferramentas, o que torna a aprendizagem de cada ferramenta um investimento não negligenciável (investimento que deve também levar em conta a vida freqüentemente efêmera das ferramentas).

Além disso, as ferramentas mencionadas anteriormente não possuem suporte suficiente à depuração de programas com grande número de processos: número de contextos dificilmente gerenciável pelo programador utilizando depuradores "clássicos", saturação das ferramentas de verificação. A combinação de uma ferramenta de depuração com uma ferramenta de visualização para fornecer uma visão de mais "alto nível" da execução das aplicações [30, 6] não permite entretanto ultrapassar os limites das ferramentas de visualização, que ainda estão aquém dos tamanhos máximos dos multiprocessadores (veja seção 3.7). O "método de depuração" utilizado classicamente consiste em se eliminar o maior número possível de erros executando o programa em um número reduzido de processadores, antes de se testar os programas em uma configuração maior.

3 Ferramentas de medida de desempenho

3.1 Introdução

A depuração do desempenho é uma fase importante no ciclo de desenvolvimento de programas paralelos, já que o motivo principal de se utilizar sistemas paralelos é a obtenção de alto desempenho. O objetivo de ferramentas de medida de desempenho é ajudar os programadores a obter o maior desempenho possível de seus programas na arquitetura utilizada. A depuração do desempenho geralmente inclui várias fases: monitoramento para recolher dados de desempenho, análise de dados para ajustar os dados brutos recolhidos e calcular índices de desempenho e a apresentação desses índices ao programador, geralmente por ferramentas de visualização sofisticadas.

O monitoramento do desempenho pode ser utilizado em pelo menos dois contextos. O primeiro ocorre quando se requer um controle dinâmico da execução do programa paralelo. Esse é o caso de algumas ferramentas que monitoram as atividades do sistema operacional, como *xload*, *perfmeter*, *top*, etc. em sistemas Unix [49], usado por administradores para gerenciar recursos computacionais. Esse é também o caso de ferramentas de monitoramento de sistemas de tempo real e de ferramentas sofisticadas de detecção automática de problemas de desempenho de programas paralelos durante sua execução, como Paradyn [38]. No segundo caso, os dados de monitoramento são analisados após o término da execução do programa (*post mortem*), visando realizar uma análise global e

mais aprofundada do comportamento do programa. Este texto trata somente desse segundo tipo de atividade de monitoramento.

Para permitir a depuração do desempenho, as ferramentas de medida de desempenho devem produzir um grande número de índices de desempenho, de forma que os programadores possam detectar e reduzir a sobrecarga existente em seus programas [3]. Esses índices podem ser divididos em duas classes principais:

- **tempos de execução**, devem ser reduzidos tanto quanto possível. O objetivo da depuração de desempenho é geralmente reduzir o tempo total de execução de um programa. Este índice pode ser decomposto em várias medidas representando o tempo gasto pela execução de diversas partes do programa (procedimentos, protocolos de comunicação, etc.).
- **taxas de utilização de recursos**, indicando o percentual da utilização dos recursos disponíveis ao programa que é gasto realizando trabalho "útil" (ou não). Se considerarmos as taxas de utilização dos processadores por exemplo, os programadores devem saber qual percentual do tempo foi gasto em várias sobrecargas, executando código de sincronização, criação de processos, término, escalonamento, comunicação, ociosidade. Taxas globais de utilização de recursos podem indicar um problema, como uma pequena utilização dos processadores ou uma grande taxa de ociosidade. Para corrigir tais problemas, freqüentemente necessita-se de dados mais detalhados. Por exemplo, uma grande taxa de ociosidade pode indicar a presença de um gargalo, cuja origem pode ser a falta de paralelismo no programa, fraco desempenho do escalonador de tarefas, uso excessivo de primitivas de sincronização, etc.

A execução de um programa paralelo pode ser monitorada em vários níveis de abstração diferentes: *hardware*, sistema operacional, ambiente de execução (*runtime*) e aplicação. Intuitivamente, o nível da aplicação é o mais significativo para os programadores de aplicações paralelas, uma vez que em geral é o único onde ele pode controlar ou ajustar parâmetros. No entanto, algumas vezes o baixo desempenho de uma aplicação pode ser explicado somente pela observação do impacto das escolhas do programador sobre o sistema de execução ou sobre o sistema operacional, durante a execução da aplicação. A capacidade de se relacionar decisões de projeto ou de programação a nível da aplicação com o comportamento do sistema de execução ou com o sistema operacional ainda é assunto de pesquisa e não será tratado neste texto.

3.2 Princípios de monitoramento de programas paralelos

A maioria das ferramentas de monitoramento são baseadas em amostragem ou orientadas a eventos [44]. Elas são descritas nas seções que seguem.

Monitoramento por amostragem A amostragem consiste em se registrar o estado do sistema observado a intervalos periódicos de tempo, geralmente por um processo inde-

pendente do processo observado. A periodicidade depende do sistema operacional (tipicamente 10 ou 20 ms em Unix). A informação registrada pode ser usada em linha (enquanto o programa se executa) ou após o término da execução do programa, para se calcular índices globais de desempenho.

`prof`, `gprof` [16] pertencem à segunda categoria: essas ferramentas calculam *post mortem* vários índices de desempenho à partir do registro da distribuição dos valores do contador de programa, realizado durante a execução do programa, e de informações sobre nomes e endereços de símbolos do programa, registrados pelo compilador. Supõe-se que o tempo gasto em um procedimento, por exemplo, seja proporcional ao número de amostras registradas com o contador de programas dentro desse procedimento.

Ferramentas de medida de desempenho baseadas em amostragem são bastante usadas para depurar o desempenho de programas seqüenciais. No entanto, esse tipo de ferramenta pode falhar em encontrar alguns problemas comuns em programas paralelos: índices globais de desempenho são de pouca ajuda para mostrar gargalos ou para avaliar o tempo gasto em comunicação ou o tempo ocioso (a não ser que haja espera ocupada—o processo utilize o processador em laço enquanto espera). Além disso, o tempo relativamente grande entre amostras pode ser inadequado para exibir fenômenos de curta duração.

Monitoramento por eventos Esta forma de monitoramento baseia-se na ocorrência de *eventos*. Parte-se do princípio que os processos que executam uma aplicação paralela produzam eventos observáveis. Um evento é definido como uma ação que troca o estado do sistema monitorado, como por exemplo uma chamada de procedimento ou a recepção de uma mensagem. A escolha dos eventos que serão observados depende do interesse do programador, e geralmente inclui emissões e recepções de mensagens, eventos que mostrem o bloqueio dos processos e eventos "definidos pelo usuário". O monitoramento por eventos associa uma data a cada evento observado. Existem diferentes formas de monitoramento por eventos: cronometragem, contagem e geração de rastros, dependendo da quantidade de informação registrada e da forma como é utilizada.

Cronometragem mede o tempo gasto nas várias partes do programa observado. Por exemplo, o tempo gasto em um procedimento pode ser obtido subtraindo-se a data de início do procedimento da data medida quando ele termina. A cronometragem requer um relógio de baixa latência. A quantidade de informação registrada é limitada a um dado por valor medido. A intrusão causada no programa depende do número de pontos de instrumentação e é potencialmente alta se uma cronometragem detalhada é requerida.

Contagem é o registro do número de ocorrências dos eventos observados em índices globais de desempenho. A contagem é geralmente considerada pouco intrusiva e gera uma quantidade reduzida de dados.

Geração de rastros é o registro de cada um dos eventos observados em um arquivo de rastros. Cada evento inclui no mínimo o tipo do evento e a data de ocorrência do mesmo.

Informação adicional também é registrada dependendo do tipo do evento. Por exemplo, se o evento registrado é uma emissão (ou recepção) de mensagem, o registro usualmente inclui o processo destinatário (ou emissor) e o tamanho da mensagem.

A geração de rastros é a técnica de monitoramento mais geral. Esta técnica pode ser usada para medir os tempos de comunicação (registrando-se os eventos de emissão e recepção) e para exibir gargalos (registrando-se onde os processos gastaram o tempo de execução). Pode também ser usada para se obter informações globais ou de contagem ou cronometragem. Por exemplo, é possível medir-se o tempo gasto na execução de um procedimento registrando-se o início e o final de cada execução desse procedimento. Por todas estas razões, a maioria das ferramentas de medida de desempenho de execuções de programas paralelos é baseada em rastros de execução [42, 19, 50, 5].

Infelizmente, a geração de rastros apresenta diversos problemas. Primeiramente, ela pode ser bastante intrusiva se o objetivo é coletar informação com alto nível de detalhe. Outro problema é que a validade da informação coletada pode ser corrompida pela interação com o sistema operacional. Por exemplo, o tempo de execução de um procedimento é a diferença entre as datas de início e fim da execução do procedimento *somente se* o processo que executa o procedimento não é suspenso durante esta execução. Portanto, todas as ferramentas baseadas na análise de rastros são bem adaptadas para a medida de desempenho de sistemas em lote, mas podem falhar em obter dados precisos em sistemas multi-usuários carregados. Esse problema pode ser diminuído utilizando-se técnicas de análise multi-níveis [40].

3.3 Geração de rastros de programas paralelos

Como definido acima, a geração de rastros registra eventos de desempenho ocorridos durante a execução de um programa em um arquivo de rastros. Como é o caso em todas as técnicas de monitoramento, a geração de rastros pode ser realizada em diferentes níveis de abstração. Existem diversas técnicas de geração de rastros: em *hardware*, em *software* ou híbridas. A qualidade dos rastros indica o nível de fidelidade da informação coletada. Esta característica é principalmente afetada pela falta de relógios globais em sistemas distribuídos, o que dificulta a ordenação de eventos que ocorreram em nós diferentes, e a intrusão devida a geração dos rastros ou efeito de sonda, que pode alterar o comportamento de programas observados em relação a programas que não o são. A qualidade dos rastros depende da técnica de geração utilizada. Esta seção fala sobre as técnicas de geração de rastros, bem como os fatores que afetam a qualidade da informação gerada.

Técnicas de geração de rastros

Em hardware: coletores de rastros deste tipo devem ser incluídos nos sistemas paralelos que se deseja observar. Tais geradores involvem o desenvolvimento de *hardware* específico e por esta razão são considerados muito custosos. A vantagem destes geradores é

que eles funcionam de forma completamente independente do programa sendo observado, não causando nenhuma alteração em seu comportamento e produzindo dados altamente confiáveis. Uma dificuldade inerente ao seu uso para o programador da aplicação é o fato de que a relação entre um evento a nível de *hardware* e uma escolha algorítmica a nível da aplicação pode não ser óbvia.

Híbrida: geradores híbridos combinam monitores específicos desenvolvidos em *hardware* com geradores a nível de *software* [24]. Como geradores a nível de *software*, geradores híbridos são acionados por instruções a nível da aplicação. É portanto mais fácil de se relacionar os eventos com as instruções que os causaram do que com um gerador a nível de *hardware*. A informação coletada é escrita em portas de *hardware* dedicadas, conectadas a *hardware* de monitoração dedicado, de forma a manter a intrusão devida ao monitoramento extremamente baixa. Adicionalmente, o *hardware* de monitoramento pode incluir um relógio global para datar os eventos. Mesmo que tais técnicas sejam consideradas ideais por programadores porque são simples de usar e produzem rastros de alta qualidade, seu uso não é muito difundido devido a problemas de falta de portabilidade e alto custo.

Em software*:* é a mais portável e a mais barata das técnicas de geração de rastros. Sua utilização pode ser realizada de forma transparente ao programador quando o gerador faz parte da biblioteca de comunicação que pode ser utilizada em modo "geração de rastros" [15]. O gerador pode igualmente ser composto por uma biblioteca que pode ser chamada pelo programa. As chamadas a essa biblioteca são inseridas pelo programador [42] ou automaticamente por um pré-processador [32]. É a técnica mais utilizada, por ser barata e relativamente fácil de ser implementada. No entanto, a obtenção de traços de alta qualidade é dificultada pela falta de relógios globais na maior parte dos sistemas distribuídos, tornando necessária a utilização de um dispositivo de *software* para gerar datas globalmente coerentes para os eventos gravados [28, 35]. Esse tipo de gerador de traços tem também problemas de intrusão no comportamento do programa analisado, uma vez que o transporte e o armazenamento dos eventos é realizado concomitantemente com a execução do programa, pelos processadores e pela rede de comunicação do sistema paralelo utilizado.

Coleta dos dados: a quantidade de dados nos rastros depende do número de eventos observados: pode ser limitada quando se observa somente os eventos de comunicações, mas pode gerar uma enorme quantidade de dados quando medições mais detalhadas são necessárias ou quando as ferramentas são mal utilizadas. Visando limitar este problema, alguns sistemas de monitoramento, como Pablo [42], ajustam automaticamente a freqüência ou a técnica de monitoramento (substituindo o registro dos eventos pelo registro de contadores de eventos) quando a freqüência dos eventos torna-se excessivamente alta. De qualquer forma, uma quantidade de dados potencialmente alta deve ser armazenada e extraída do sistema paralelo. Vários compromissos podem ser considerados entre a sobrecarga na

utilização de memória, com alocação de grandes *buffers* para conter os eventos, e a sobrecarga em tempo, resultante da utilização de algoritmos de compressão [34] ou do tempo gasto na transferência dos dados para disco.

Formato dos rastros: não existe um consenso na comunidade científica sobre a possibilidade ou a necessidade da definição de um formato padrão de arquivos de rastros de execução. A utilização de formatos de rastros auto-definidos como SDDF utilizado pela ferramenta Pablo [42] é um conceito bastante poderoso: a estrutura dos registros de eventos é definida nos cabeçalhos do próprio arquivo. Um outro formato bastanto utilizado é PICL, por ser o formato da ferramenta ParaGraph [19], bastante popular. A conversão entre formatos de arquivos representa um problema somente quando não se consegue exprimir algum conceito no formato destino (o formato PICL não prevê que se possa identificar fluxos de execução além de processos, por exemplo), o que pode resultar em perda de dados ou adaptações nos mesmos para que possam ser visualizados por uma dada ferramenta.

Qualidade da informação nos rastros Idealmente, os eventos registrados seriam datados por um relógio global de precisão infinita e não causariam nenhuma intrusão no programa observado. No entanto, este não é o caso em geral, e especialmente no caso da utilização de técnicas de geração de rastros por *software*. Eventos datados com relógios locais diferentes podem ser incoerentes entre si. A intrusão devida à captura dos rastros pode alterar o comportamento da execução do programa observado.

Qualidade da medida do tempo Em sistemas distribuídos, cada processador tem seu próprio relógio físico. Essa falta de um relógio global pode resultar em incoerências entre os eventos registrados independentemente por cada processador usando seus relógios locais. Por exemplo, a data registrada no evento de recepção de uma mensagem pode ser inferior à data registrada para seu envio. Essas incoerências dificultam ou impossibilitam a análise dos rastros de execução por ferramentas de medida de desempenho. Em geradores de rastros em *hardware* ou híbridos, esse problema é resolvido pelo uso de *hardware* dedicado [23]. Em geradores em *software*, esse problema pode ser atacado pela implementação de algoritmos de correção de relógio (seção 3.5).

Usando as propriedades físicas dos osciladores de quartzo comumente utilizados nos relógios dos computadores, é possível modelar a hora local $lt_i(t)$ medida no processador i como uma dependência linear [35]:

$$lt_i(t) = \alpha_i + \beta_i t + \delta_i, \quad i \in [1, p], \tag{1}$$

onde t representa o tempo "absoluto" ou "universal", a constante α_i é a diferença do relógio i no tempo $t = 0$, a constante β_i (próxima a 1) é o escorregamento do relógio i com relação a t, e a variável aleatória δ_i modela a granularidade e outras perturbações aleatórias. Pode-se considerar que δ_i seja independente do tempo t. Este modelo é correto

somente se os parâmetros físicos (temperatura, por exemplo) do ambiente (sala dos computadores) permanecem constantes e o tempo total da experiência seja suficientemente pequeno para se poder desconsiderar o envelhecimento dos cristais. Se essas restrições não são satisfeitas, os coeficientes α_i e β_i podem não mais ser constantes.

Intrusão causada pela geração de rastros Como qualquer técnica de monitoramento, a geração de rastros perturba a execução dos programas paralelos observados. É difícil estimar a intrusão porque ela depende do programa observado e do número de eventos registrados. No caso de geração em *hardware* ou híbrida, sabe-se que a intrusão é limitada a um percentual bem baixo do tempo de execução e que ela não altera o comportamento do programa o suficiente para inviabilizar sua utilização por ferramentas de medida de desempenho [23].

A intrusão não pode ser negligenciada no caso da geração de rastros em *software*. Várias propostas foram feitas para modelar e compensar essa intrusão [36, 50, 33, 34]. Três objetos estão envolvidos no processo de modelagem e compensação:

1. o arquivo de rastros T, refletindo um comportamento perturbado da aplicação;
2. o rastro de uma execução "ideal" T_0, que seria obtido por uma instrumentação ideal, não intrusiva;
3. o rastro aproximado T_a, obtido pela aplicação de um modelo de compensação de intrusão em T.

No caso da geração de rastros por *software*, na falta de um monitor de *hardware* não intrusivo, o único índice de desempenho que pode ser conhecido de T_0 é seu tempo de execução. A importância da intrusão da geração de rastros por *software* em T com a relação a T_0 pode ser avaliada comparando-se os respectivos tempos de execução. Quando modela-se essa intrusão, dois tipos de perturbação são em geral definidos [36]:

Perturbação direta: resultante da execução pelos processos instrumentados das instruções adicionais para se gerar os rastros — tempo gasto lendo o relógio e construindo a descrição de um evento em memória — e para armazenar o rastro em arquivos.

Perturbação indireta: localizada fora do código de instrumentação mas causada por esse código. O monitoramento pode de fato alterar a forma como os processos são escalonados e a memória é referenciada (freqüencia de faltas de páginas e faltas na memória *cache*). Pode também prejudicar algumas otimizações de compilação e penalizar o desempenho do subsistema de E/S, incluindo o sistema de arquivos e a rede de comunicações.

Modelos de compensação de perturbação não levam em conta as perturbações indiretas, uma vez que estas perturbações não podem ser estimadas no nível de abstração da aplicação. No entanto, existem um grande esforço a fim de para limitar os fatores que influenciam as perturbações indiretas, como o volume de dados gerados.

3.4 Dificuldades advindas da utilização de múltiplos fluxos de execução

A geração de rastros concerne essencialmente o escalonamento dos fluxos de execução, ou seja, para cada fluxo necessita-se registrar a sua identidade, o início, o final e a causa de cada período de suspensão de sua atividade. Acrescenta-se a isto informações que permitam reconstituir o histórico das comunicações.

A multiprogramação de cada nó de processamento traz problemas de identificação dos fluxos de execução, observabilidade do seu escalonamento, atomicidade dos eventos e gerência dos *buffers* de rastros.

Identificação dos fluxos de execução É conveniente que os fluxos de execução que compartilham um nó multiprogramado sejam identificados. Os núcleos de multiprogramação leve (*threads*) geram tais identificadores, não permitindo infelizmente sua utilização para a construção de ferramentas portáveis de geração e visualização de rastros de execução. A norma Posix [27], por exemplo, especifica o nome do tipo do identificador, mas não sua representação interna, que é dependente de implementação.

A possibilidade de se associar uma variável estática privada aos fluxos de execução (*get* e *set specific* na norma POSIX) permite se contornar o problema de representação. O endereço dessa variável pode servir de identificador de fluxo de execução, representando este endereço como um valor inteiro.

Identificação dos correspondentes As bibliotecas de comunicação como PVM e MPI indicam o nó de origem das mensagens, o que permite encontrar a correspondência entre os eventos de envio e de recepção de mensagens em arquivos de rastros gerados em nós diferentes. Quando os nós são multiprogramados, somente a identificação do nó emissor torna-se insuficiente: a identificação do fluxo emissor necessita uma informação complementar.

Para se gerar rastros das comunicações com multiprogramação necessita-se intervenir sobre a biblioteca de comunicação para retirar a ambiguidade dos rastros sobre o fluxo emissor. Duas estratégias podem ser consideradas:

- adicionar uma identificação do fluxo emissor às informações (como o nó emissor) transportadas junto ao conteúdo da mensagem. Esta solução em geral implica na alteração do código fonte da biblioteca de comunicação;
- serializar e numerar em cada nó os pedidos de comunicação. Se as mensagens são enviadas na mesma ordem que são postadas (propriedade FIFO do núcleo de comunicação), a cronologia das submissões permite escolher a boa associação *post mortem*.

Observabilidade do escalonamento A atividade de um único fluxo de execução em um nó sequencial é fácil de rastrear: ele só pode se bloquear através de primitivas explícitas de comunicação, sendo suficiente registrar-se o início e o final de tais primitivas. Com a

multiprogramação, um fluxo de execução pode também ter seu progresso suspenso implicitamente, por decisão do escalonador, que entrega o processador a outro fluxo. Isso conduz naturalmente ao registro da atividade do escalonador (transições entre os estados ativo, pronto e bloqueado de cada fluxo).

O registro de tais eventos esbarra em dois obstáculos: a disponibilidade de núcleos de multiprogramação leve instrumentados — um problema geral de observação quando o comportamento do nível do sistema que se está observando depende do comportamento de níveis subjacentes que não são obrigatoriamente observáveis — e a freqüência das transições (problema de intrusão e de tamanho dos rastros). O resultado dessas restrições é que, em geral, os rastros informam somente os períodos de bloqueio dos fluxos de execução, e a repartição do tempo do(s) processador(es) entre os diferentes fluxos aparentemente ativos a cada instante somente poderá ser deduzida dos rastros sob hipóteses restritivas particulares (como o conhecimento do comportamento do escalonador).

Atomicidade e datação dos eventos A multiprogramação causa um problema clássico de atomicidade capaz de falsificar as datas dos eventos. O registro de um evento é realizado por uma seqüência de operações (reserva de espaço em *buffer*, leitura da data do sistema, chamada da primitiva a instrumentar, escrita do evento no *buffer*), que pode ser interrompida pelo escalonador para ceder o processador a outro fluxo.

Resulta desse fato uma incerteza sobre a datação dos eventos, uma vez que toda comutação de fluxo de execução entre a leitura da data e a chamada da primitiva causará o registro de uma data falsa para o evento correspondente a essa primitiva (essa incerteza reflete a inobservabilidade do escalonamento dos fluxos de execução.

Gerência de *buffers* Os rastros são tipicamente armazenados em um *buffer* de memória, mais tarde copiado para disco (quando o *buffer* enche ou quando o programa termina). O compartilhamento por vários fluxos de um *buffer* de rastros comum aumenta a intrusão devida à geração de rastros, porque implica uma exclusão mútua a cada acesso (problema clássico de múltiplos escritores), o que afeta o escalonamento dos fluxos de execução. A exclusão mútua afeta também a exploração do paralelismo interno a nós multiprocessadores.

Dotar os fluxos de execução de *buffers* individuais substitui o problema de compartilhamento pelo problema de dimensionamento dos *buffers*: alguns fluxos terão uma vida efêmera enquanto outros gerarão rastros de grande tamanho.

O compartilhamento de uma coleção de blocos de memória de tamanho fixo alocados por demanda constitui um compromisso interessante entre as duas alternativas. Um fluxo não mais realiza um acesso à estrutura compartilhada a cada evento, mas necessita alocar um *buffer* suplementar cada vez que seu *buffer* corrente enche. Os *buffers* cheios podem ser reciclados por um fluxo adicional (um "*daemon*"), após cópia de seus conteúdos no arquivo de rastros.

3.5 Implementação de relógio global em sistemas paralelos com memória distribuída

Como mencionado anteriormente, vários sistemas distribuídos não possuem um relógio global. O uso de relógios locais para datar os eventos resulta em erros onde a ordem dos eventos, derivada de suas datas, pode contradizer a relação causal entre esses eventos. Para se evitar estes erros, é possível implementar-se uma data global em um sistema distribuído selecionando-se o relógio de um dos processadores como o relógio de referência [35]. A equação 1 pode ser derivada em:

$$lt_i(t) = \alpha_{i,r} + \beta_{i,r} lt_r(t) + \delta_{i,r}. \quad (2)$$

A data global corrigida no processo i, chamada $LC_i(t)$ pode então ser estimada como:

$$LC_i(t) = lt_r(t) \approx \frac{lt_i(t) - \alpha_{i,r}}{\beta_{i,r}}, \quad (3)$$

o valor $lt_r(t)$ na equação 3 sendo o valor do relógio de referência no tempo t, como ele pode ser calculado de $lt_i(t)$, desde que $\alpha_{i,r}$ e $\beta_{i,r}$ sejam conhecidos. Esses coeficientes têm que ser estimados para cada um dos processadores do sistema. A forma de estimá-los é através de amostragens estatísticas das datas que alguns eventos que ocorrem no processador de referência (cujas datas são medidas com o relógio de referência) teriam no processador i. Os eventos usados para essas estimativas são mensagens de "ping-pong" trocadas entre o processador de referência e cada um dos demais processadores antes e/ou após a execução do programa (veja [34] para mais detalhes).

3.6 Interação com o sistema operacional

A geração de rastros é uma técnica bem adaptada à captura de fenômenos que ocorrem no nível da aplicação, como tempos de comunicações. Ela pode no entanto falhar na captura de ineficiências em sistemas multi-usuários carregados. O motivo é que esta técnica não permite distinguir entre os estados *ativo* e *ativável* de um processo: apenas eventos explícitos de bloqueio dos processos, em nível de aplicação, são registrados. O processo em questão pode ser suspenso pelo sistema operacional sem que isso tenha nenhuma relação com em evento no nível da aplicação. Nesses casos, a duração medida para algumas atividades do processo excederá a duração real de tais atividades. Tal fenômeno pode ser detectado conjugando-se os rastros gerados no nível da aplicação com dados capturados ao nível do sistema operacional [40].

3.7 Visualização

Devido à complexidade das execuções de programas paralelos, a utilização de sistemas de visualização sofisticados é a técnica de apresentação mais freqüentemente utilizada. O "motor" das ferramentas de visualização é um simulador[1] dirigido pelos rastros, que

[1] Mesmo sendo um simulador, os resultados produzidos são "reais", já que foram medidos durante uma execução "real".

reconstitui a atividade dos processadores e as comunicações. A apresentação gráfica dos resultados dessa simulação tem por objetivo facilitar sua compreensão, já que a quantidade de informação apresentada pode ser bastante significativa.

Representações clássicas Existem várias formas de se representar gráficamente a atividade dos processadores e as comunicações que ocorreram durante a execução dos programas observados (alguns exemplos podem ser vistos na figura 3). Além dessas informações, a maioria das ferramentas de visualização calculam e apresentam informações globais de desempenho. A maioria das representações existentes foram integradas na ferramenta de visualização ParaGraph [19], que por esta razão e por sua portabilidade foi bastante difundida. A implementação de ParaGraph é baseada em padrões (X-Windows, Unix), o que garantiu sua portabilidade para a maioria das estações de trabalho. Os rastros de execução são coletados de forma transparente ao programador pela biblioteca de comunicação instrumentada PICL [15], ou por outros geradores de rastros e posteriormente convertidos no formato de PICL [33] para poderem ser visualizados por ParaGraph. Posteriormente, outras representações visuais mais elaboradas (inclusive com utilização de gráficos em três dimensões) foram propostas [17].

Devido às diferentes representações e em particular às representações da atividade dos processadores em função do tempo, por processador (diagrama de Gantt) e acumulada (diagrama de Kiviat), assim como às representações das comunicações (diagrama "espaço-tempo"), é possível se evidenciar certos gargalos provenientes por exemplo de um mau recobrimento do tempo de comunicação por processamento útil, e de eliminá-los [20]. De todas as representações disponíveis nas ferramentas existentes, o diagrama espaço-tempo e sua variante "de Hasse" parecem ser as mais úteis à compreensão dos programas que utilizam paralelismo de controle.

A correlação entre as visualizações da execução e o programa fonte representam por vezes um problema difícil. Este, em geral, não é o caso para um programa que utiliza passagem de mensagens explícitas e em número limitado, mas pode sê-lo quando o número de mensagens é grande e existem mensagens geradas implicitamente, sem relação aparente com o programa fonte. Algumas ferramentas interativas oferecem a oportunidade de se "interrogar" as entidades representadas para se obter mais informação sobre elas, como a porção do código que as causou, ou o volume ou mesmo os próprios dados que foram enviados em uma comunicação [50]. Como no caso dos programas que se utilizam do paralelismo de dados (próximo parágrafo), essa relação nem sempre é suficiente para explicar, em um dado nível de abstração, os fenômenos que são observáveis somente em um nível diferente. Por exemplo, no caso em que a execução de um "*daemon*" do sistema operacional —observável a nível sistema— perturba a execução de um processo do usuário —observado normalmente a nível aplicativo— em um processador da máquina.

Caso do paralelismo de dados Os processos utilizados pelos programas que exploram o paralelismo de dados executam as mesmas instruções, de maneira mais (SIMD) ou me-

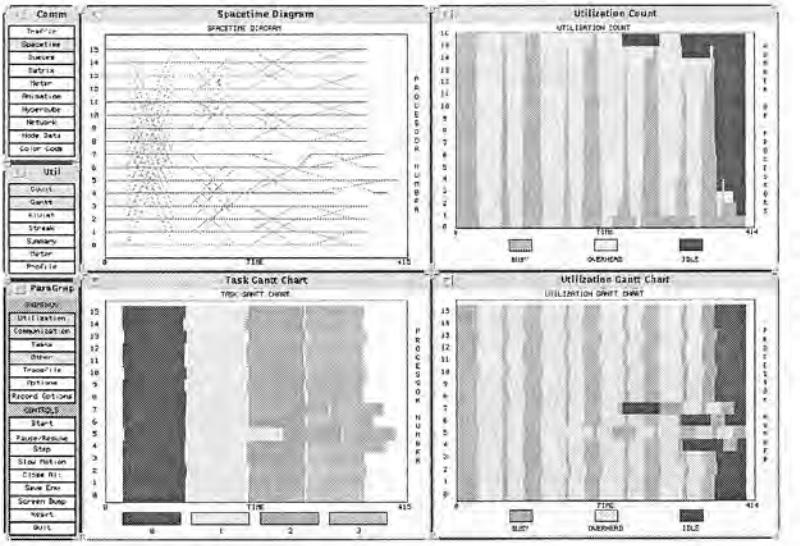

Figura 3. Exemplos de visualizações de ParaGraph

nos (SPMD) síncrona, sobre dados diferentes. O desempenho desses programas depende essencialmente da forma como os dados são distribuídos entre os processadores utilizados. As visualizações das execuções desses programas [48, 5, 18] mostram a evolução dos dados manipulados, geralmente matrizes multidimensionais de grande tamanho. Essas visualizações são utilizadas tanto para a depuração —que não difere muito da depuração seqüencial nesse tipo de aplicação— quanto para a otimização. No entanto, a otimização desse tipo de programa é difícil porque os programas executados pelas máquinas paralelas não correspondem ao nível de abstração manipulado pelos programadores. O compilador gera automaticamente as comunicações, a partir da distribuição dos dados descrita pelo programador. Se essas comunicações se revelam ineficientes, não há muito que o programador possa fazer.

Limitações das ferramentas de visualização Apesar da sua sofisticação, as ferramentas existentes sofrem um certo número de limitações [44]. Falaremos apenas de duas: a falta de escalabilidade e a dificuldade de se representar dados multidimensionais.

Ainda que a maioria das representações oferecidas pelas ferramentas como ParaGraph permitam visualizar a atividade de até 128 ou mesmo 512 processadores (processos), es-

sas representações tornam-se incompreensíveis com algumas dezenas de processadores. O tamanho de uma tela impõe, de qualquer forma, um limite físico no número máximo de processos observáveis: é impossível representar uma matriz de comunicação com várias centenas de processos por falta de pixels disponíveis. Esse problema se agrava para os modelos de programação hierárquicos nos quais vários fluxos de execução estão em atividade no interior de cada processo [12, 4, 14], sendo esses fluxos provavelmente criados e destruídos dinamicamente no decorrer da execução do programa.

Uma possibilidade de ataque a esse problema é utilizar-se uma representação hierárquica da execução paralela: grupos de processadores, processos de um grupo, fluxos de execução no interior de um processo. A definição de grupos pode ser feita pelo usuário em função de sua aplicação. A figura 4 mostra uma visualização contendo processadores agrupados.

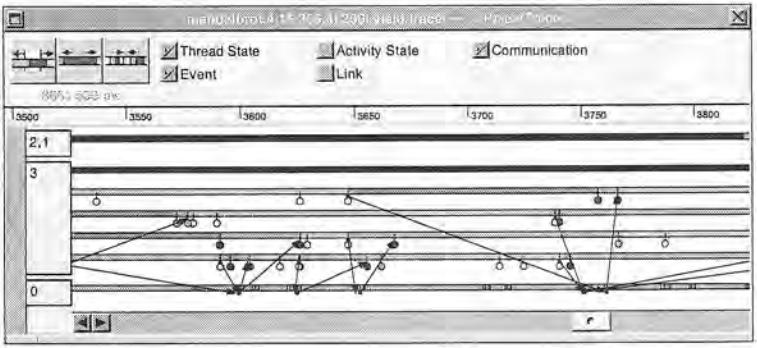

Figura 4. Visualização com grupo de nós em Pajé

Uma outra possibilidade seria utilizar técnicas de agrupamento dinâmico estatístico dos dados para reduzir o número de informações a apresentar ao usuário, que teria acesso aos agregados mais pertinentes evidenciados pela busca automática de gargalos no programa [44].

O desempenho de programas depende de um número de parâmetros freqüentemente superior às três dimensões dos sistemas gráficos clássicos. A representação e a navegação nestes espaços n-dimensionais é um problema aberto de pesquisa, tendo sido propostos (e implementados!) ambientes de realidade virtual para explorá-los [43].

Uma forma complementar de se representar o comportamento temporal de um programa paralelo consiste na utilização de sons, normalmente sintetizados devido à quantidade

de tratamentos que se pode realizar sobre eles. A sonorização é utilizada, como no cinema, de forma complementar à visualização. O tratamento do som por um observador pode ser realizado de forma passiva, sendo por vezes mais fácil que a visualização para a detecção de anomalias, principalmente em esquemas repetitivos. Os sons têm ainda a capacidade de poderem se combinar, o que é interessante para representar o comportamento de diversas entidades em paralelo. O som apresenta as seguintes dimensões, sobre as quais se pode jogar para se passar uma informação: timbre, altura, duração, intensidade, posicionamento espacial. As experiências relatadas na literatura falam essencialmente sobre a representação sonora de comunicações ou da carga dos processadores.

4 Conclusão

A depuração de programas paralelos eficientes sempre foi considerada como um exercício difícil e por isso suscitou uma grande quantidade de trabalhos. As ferramentas de auxílio à depuração têm o objetivo de ajudar os programadores a identificar os erros de desempenho de seus programas. Essas ferramentas apresentam uma grande sensibilidade à intrusão e devem representar uma grande quantidade de dados de uma forma inteligível e coerente.

Um grande número de ferramentas são baseadas na geração de rastros de execução de aplicações paralelas, sendo que o método mais utilizado para gerá-los é através de geradores por *software*, mais simples e mais portável que os demais. Esse método necessita um dispositivo de datação global, que pode eventualmente ser implantado também em *software*. Seu principal inconveniente é a intrusão que causa na execução dos programas, mesmo que existam técnicas que permitam de se avaliar e corrigir essa intrusão em vários casos. Apesar da utilização de ferramentas de visualização (ou mesmo de sonorização) bastante sofisticadas, a representação de execuções paralelas encontra rapidamente seus limites quando o número de processos aumenta.

Pode-se finalmente notar que, de uma forma análoga à depuração clássica, as ferramentas de ajuda à depuração do desempenho são relativamente pouco utilizadas, sem dúvida pelas mesmas razões: multiplicidade de modelos de programação e falta de profissionalismo de várias ferramentas.

Referências

1. BBN Systems and Technologies. *TotalView multiprocess debugger. User's Guide, version 3.4*, 1995.
2. L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors. *Proc. Euro-Par'96 Parallel Processing*, number 1123 in LNCS. 1996.
3. J.M. Bull. A hierarchical classification of overheads in parallel programs. In *Proceedings of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219. Chapman Hall, 1996.
4. M. Christaller, J. Briat, and M. Rivière. Athapascan-0 : concepts structurants simples pour une programmation parallèle efficace. *Calculateurs Parallèles*, 7(2):173–196, 1995.
5. C. Clémençon, A. Endo, J. Fritscher, A. Müller, and V.J.N. Wylie. Annai scalable run-time support for interactive debugging and performance analysis of large-scale programs. In Bougé et al. [2].

6. C. Clémençon, A. Endo, J. Fritscher, A. Müller, and V.J.N. Wylie. Annai scalable run-time support for interactive debugging and performance analysis of large-scale programs. Technical Report TR-96-04, Swiss Center for Scientific Computing, CSCS/SCSC, Zürich, Suisse, 1996. Serveur WWW : http://www.cscs.ch/.
7. C. Clémençon, J. Fritscher, M.J. Meehan, and R. Rühl. An implementation of race detection and deterministic replay with MPI. In S. Haridi, K. Ali, and P. Magnusson, editors, *Euro-Par'95 Parallel Processing*, volume 966 of *LNCS*, pages 155–166. Springer-Verlag, August 1995.
8. Hewlett Packard Company. *HP/DDE Debugger User's Guide*, July 1996. B3476-90015.
9. Cray Research, Inc. *UNICOS Symbolic Debugger Reference Manual*, June 1991. SR-2091 6.1.
10. S. K. Damodaran-Kamal. *Testing and Debugging nondeterministic message passing parallel programs*. PhD thesis, University of Southwestern Louisiana, USA, 1994.
11. A. Fagot and J. Chassin de Kergommeaux. Formal and experimental validation of a low-overhead execution replay mechanism. In S. Haridi, K. Ali, and P. Magnusson, editors, *Euro-Par'95 Parallel Processing*, volume 966 of *LNCS*, pages 167–178. Springer-Verlag, August 1995.
12. I Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.
13. J. M. Francioni and C. M. Pancake. High Performace Debugging Standards Effort, 1998. `http://www.ptools.org/hpdf`.
14. J.-M. Geib, F. Hémery, J.-F. Méhaut, J.-F. Roos, and E.-G. Talbi. PVC : un environnement de programmation parallèle avec régulation de charge. *Calculateurs Parallèles*, 7(2):139–158, 1995.
15. G. A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. PICL, a portable instrumented communication library. TN 37831-8083, Oak Ridge National Laboratory, Oak Ridge, USA, 1991.
16. S. Graham, P. Kessler, and M. McKusik. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, pages 120–126. ACM, 1982.
17. S.T. Hackstadt and A.D. Malony. Next-generation parallel performance visualization. Technical report CIS-TR-93-23, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, USA, October 1993.
18. S.T. Hackstadt and A.D. Malony. Distributed array query and visualization for High Performance Fortran. In Bougé et al. [2].
19. M. T. Heath and J. E. Finger. ParaGraph: A Tool for Vizualizing Performance of Parallel Programs. Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, Oak Ridge, TN, 1991.
20. M.T. Heath. Recent Developments and Case Studies in Performance Visualization using ParaGraph. In *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.
21. J.-M. Hélary, A. Mostefaoui, and M. Raynal. Déterminer un état global dans un système réparti. Technical Report 2090, INRIA Rennes, November 1993.
22. D.P. Hembold and C.E. McDowell. Race detection — ten years later. In Simmons et al. [47], pages 101–126.
23. R. Hofmann. Monitoring and evaluation of parallel and distributed systems. In J. Chassin de Kergommeaux and D. Trystram, editors, *Proc. European School on Parallel Programming Environments, ESPPE'96*, pages 135–153. Institut d'Études Scientifiques Avancées de Grenoble, 1996. Accessible par ftp://ftp.imag.fr/pub/APACHE/ESPPE96.
24. R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance monitoring: Methods, tools, and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.
25. R. Hood. The P2D2 project: Building a portable distributed debugger. In *Proceedings of the SIGMETRICS Symposium on Paralel and Distributed Tools*, Philadelphia, May 1996.
26. IBM. *IBM Parallel Environment for AIX: Operation and Use, Volume 2, Tools Reference Version 2, Release 2*. Document Number SC28-1980-00.
27. IEEE Standard for Multithreaded Programming. *Posix 1.c Standard*. IEEE Press, 1995.
28. J.-M. Jézéquel. Building a global time on parallel machines. In *Proc. of the 3rd International Workshop on Distributed Algorithms*, number 392 in Lecture Notes in Computer Science, pages 136–147. Springer Verlag, 1989.
29. T.J. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.
30. E. Leu and A. Schiper. Execution replay : a mechanism for integrating a visualization tool with a symbolic debugger. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *CONPAR 92 - VAPP V. Second Joint International Conference on Vector and Parallel Processing*, volume 634 of *Lectures Notes in Computer Science*. Springer-Verlag, 1992.

31. Eric Leu. *La réexecution, pierre angulaire de la mise au point des programmes parallèles*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, July 1992.
32. É. Maillet. Tape/pvm: An efficient performance monitor for pvm applications. User guide, LMC-IMAG, B.P. 53, F-38041 Grenoble Cedex 9, France, 1994. Available at ftp://ftp.imag.fr/imag/APACHE/TAPE.
33. É. Maillet. Issues in Performance Tracing with Tape/PVM. In *Proceedings of EuroPVM'95*, pages 143–148. HERMES (ISBN 2-86601-497-9), 1995.
34. É. Maillet. *Traçage logiciel d'applications parallèles : conception et ajustement de qualité*. PhD thesis, Institut National Polytechnique de Grenoble, September 1996.
35. É. Maillet and C. Tron. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 28:84–93, July 1995.
36. A. D. Malony, A. Reed, and H.A.G. Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on parallel and distributed systems*, 3(4), July 1992.
37. F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.
38. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, November 1995.
39. R.H.B. Netzer and B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *Proceedings of Supercomputing '92*, Minneapolis, Minnesota, November 1992. Institute of Electrical Engineers Computer Society Press.
40. F.-G. Ottogali and J.-M. Vincent. Mise en cohérence et analyse de traces multi-niveaux. Submitted to publication. In French.
41. C.M. Pancake. Collaborative efforts to develop user-oriented parallel tools. In Simmons et al. [47], pages 355–366.
42. D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz. An Overview of the Pablo Performance Analysis Environment. Technical report, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1992.
43. D.A. Reed, K.A. Shields, W.H. Scullin, L. F. Tavera, and C.L. Elford. Virtual reality and parallel systems performance analysis. *IEEE Computer*, November 1995.
44. Daniel A. Reed. Experimental analysis of parallel systems: Techniques and open problems. In G. Haring and G. Kotsis, editors, *Proc. 7th Int. Conference on Computer Performance Evaluation*, Vienna, Austria, May 1994. Springer Verlag.
45. Michiel Ronsse and Koen De Bosschere. JiTI: Tracing Memory References for Data Race Detection. In *Parallel Computing: Fundamentals, Applications and New Directions*. Elsevier, 1997.
46. Silicon Graphics, Inc. *CASEVision/Workshop User's Guide*, April 1992. 007-1523-020 and 007-1524-020.
47. M.L. Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, editors. *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Society, 1996.
48. S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title. *Data Visualization and Performance Analysis in the Prism Programming Environment*, pages 37–52. Elsevier Science Publishers B.V. (North Holland), 1992.
49. W.R. Stevens. *Unix network programming*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
50. J. C. Yan. Performance tuning with AIMS — an automated instrumentation and monitoring system for multicomputers. In *Proc. of the Twenty-Seventh Annual Hawai Conference on System Sciences*, pages 625–633. IEEE Computer Society Press, 1994.