Universidade de São Paulo
BRASIL

Instituto de Ciências Matemáticas e de Computação

Teste e Depuração de Programas Paralelos

SCE 217 - Programação Concorrente

Professora Doutora Regina Helena Carlucci Santana

Alunos:

Fábio Lima Santos 3280979

Fernando Vacari 3341285

Juliano Rosa 3309340

São Carlos, junho de 2003

Índice

1. Introdução

"First, the bad news. Adding printf() calls to your code is still a state-of-the-art methodology."

Extraído do site http://www.netlib.org/pvm3/book

Grande parte dos artigos que discursam sobre depuração de programas paralelos começa com a mensagem (implicitamente ou explicitamente) de que, enquanto a depuração de programas seqüenciais pode ser caracterizada como uma arte difícil (difficult art), a depuração de um programa paralelo é uma obrigação dolorosa (painful chore). Isto por que a quantidade de problemas (ainda sem soluções definitivas) relacionados com a depuração e teste de programas paralelos é intratável para se fazer uma depuração 100% livre de erros. Enquanto na programação seqüencial temos erros como loops infinitos, incompatibilidade de tipos, problemas com ponteiros e memória, etc, na programação paralela temos todos estes problemas agravados e mais alguns acrescentados.

A busca por soluções nesta área tem sido intensa, porém ainda há muito a ser desenvolvido. Esta busca tem se dividido em duas frentes de pesquisa e desenvolvimento. A primeira abordagem está em desenvolver ferramentas para depuração e teste que suportem modelos de programação paralela existentes e difundidos. A segunda abordagem está em desenvolver um modelo ou arquitetura para desenvolvimento de programas paralelos que tenta evitar que o programador cometa erros.

Apesar de todos os esforços de desenvolvimento de um paradigma de depuração e testes de programas paralelos, o que se tem conseguido são ferramentas específicas para certas plataformas e que ainda possuem diversos problemas e debilidades na execução de sua tarefa. Além disso, estas ferramentas se encontram (a maioria) em fase puramente experimental.

Durante este trabalho discursaremos sobre teste de programas paralelos, seus problemas e técnicas existentes. Depois serão expostas algumas técnicas para a depuração de programas paralelos que contém erros. Posteriormente apresentaremos algumas

ferramentas e modelos para o teste e depuração de programas paralelos. Finalmente abordaremos alguns modelos de programação que visam evitar a ocorrência de erros.

2. Teste de Programas Paralelos

Existe uma regra em computação que pode não ser conhecida de todos os programadores, mas que certamente todos eles já a observaram pelo menos uma vez: “Todo programa não trivial contém erro”. Por isso, a execução bem sucedida de testes é um elemento crítico fundamental de garantia de qualidade de software. Assim, podemos definir teste como um método que serve para se descobrir se existem erros em um programa. Erros são quaisquer ocorrências ou eventos que venham a fazer com que o software não apresente o comportamento ou resultado desejado.

Quando falamos em programas seqüenciais, existe uma grande diversidade de métodos de teste já consolidados e mais uma grande quantidade de pesquisa na área. Porém, ainda existe um percurso muito grande a ser percorrido pelos pesquisadores. Certamente esta é a área menos desenvolvida em termos de ferramentas para auxílio ao desenvolvimento de software. Entretanto, nesta área já existem bastantes estudos teóricos baseados em modelos matemáticos e estatísticos, visando aprimorar, de forma eficiente, a detecção de erros em programas seqüenciais.

Já em programas e algoritmos paralelos o desenvolvimento e a pesquisa têm esbarrado em problemas pouco explorados ainda, sendo que as abordagens seqüenciais são, em geral, ineficazes neste novo paradigma. Para se poder entender melhor a dimensão do problema no teste de programas paralelos, é preciso saber quais são os erros que podem ocorrer quando se executa código concorrentemente.

Erros

Programas paralelos apresentam os mesmos tipos de erros de programas seqüenciais e mais alguns devido à concorrência e sincronismo entre o código que está sendo executado

concorrentemente. Assim, problemas como loops infinitos ou manipulação de memória (ponteiros) ainda persistem em programas paralelos.

Devido à concorrência, alguns novos problemas costumam aparecer em programas paralelos: race condition, deadlocks e livelocks.

Race Condition

Race condition (condição de corrida) refere-se à situação em que dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e o resultado final depende de quando precisamente cada processo foi executado.

Deadlock

Deadlock é a situação em que dois ou mais processos que estão interagindo e em algum momento podem entrar em um estado em que não podem continuar seu processamento por estarem aguardando uma resposta ou uma condição que nunca será atingida. Nesta situação ocorre o “travamento” dos processos.

Livelock

Similar ao Deadlock, no entanto, os processos não são travados, eles continuam executando uma parte de seus códigos, mas nunca conseguirão obter o resultado pretendido.

O grande problema na identificação de erros em programas paralelos está no fato de que este tipo de programa não é determinístico. Em um programa seqüencial, dada uma entrada espera-se uma saída correspondente (correta ou não). Ao se repetir o processo no mesmo programa, ou seja, aplicar a mesma entrada, se obterá a mesma saída obtida anteriormente.

Já em um programa paralelo, uma entrada pode produzir saídas diferentes, dependendo do estado do sistema durante a execução do programa. Isto geralmente ocorre

em programas que contém erros, porém, o problema está em se determinar em quais estados do sistema o programa não está se comportando corretamente.

Como resultado do não determinismo em programas paralelos, ocorre um fenômeno chamado Efeito de Intrusão, que resulta do fato de a observação de um programa paralelo poder afetar o comportamento do mesmo, ou seja, a tentativa de ganhar mais informações sobre o programa pode alterar o comportamento e o estado do sistema. Por exemplo, se um programa paralelo apresenta um erro de sincronismo, a simples inserção de uma linha de verificação ou a própria ação do depurador pode influenciar no comportamento do programa, sincronizando as tarefas.

Basicamente testar um sistema consiste em se desenvolver um caso de teste, ou seja, desenvolver uma entrada e uma saída correspondente ao resultado esperado pelo sistema. O software é então testado com a entrada determinada e compara-se a saída obtida com a saída esperada. Em programas concorrentes, como vimos anteriormente, o resultado obtido não é determinístico, por isso, são necessários vários testes com a mesma entrada, mudando o estado do sistema e, de alguma forma, controlando o comportamento do paralelismo do software a ser testado, visando se alcançar o máximo de casos de teste possíveis.

Como é difícil se determinar quais são todos os possíveis estados do sistema, e ainda mais difícil reproduzir cada um deles, existem ferramentas específicas que tentam efetuar de forma automática o controle do comportamento e a realização de testes em um software.

Existe ainda mais um problema que pode ocorrer (e na verdade ocorre com relativa freqüência) na realização de testes. É fácil de se observar que um programa não precisa ser obrigatoriamente determinístico. Existem sistemas que devem realmente produzir um comportamento ou resultado diferente de acordo com o estado do sistema, ou seja, pode ser que o programador tenha inserido algo que pareça um erro, mas que na verdade é uma funcionalidade do sistema e é completamente conhecida e controlada pelo desenvolvedor.

3. Depuração de Programas Paralelos

A depuração de programas paralelos é uma conseqüência de testes bem sucedidos. Ou seja, programas paralelos são depurados quando existem erros em seu funcionamento.

Em geral, quando tratamos de depuração de programas paralelos, estamos falando no processo de descoberta da localização de um erro. Assim, o teste determina que existe um erro, para determinada entrada e estado do sistema, enquanto a depuração determina onde, no código, este erro está ocorrendo.

Devido ao seu caráter abstrato no tratamento de erros em algoritmos, a depuração chega a ser considerada uma arte, e não uma ciência. Não existem métodos prontos para se descobrir onde e porque um código não está funcionando. Existem técnicas e ferramentas que auxiliam o programador na execução desta tarefa.

3.1 Depuração Tradicional

A depuração tradicional consiste em existir um depurador para cada processo. Cada processo é então analisado separadamente e concorrentemente, existindo algumas técnicas para o funcionamento de cada depurador:

- Depuração de Saída: consiste em se exibir, passo a passo, o valor das variáveis e o estado do sistema. Este método é amplamente utilizado, apresentando resultados satisfatórios em programas não muito complexos.
- Tracing: permite que o valor das variáveis seja visualizado em certos pontos da execução do programa.
- Breakpoints: permite que em determinado ponto da execução de um programa o valor das variáveis seja visualizado e modificado.
- Controle de execução: permite determinar a ordem de execução dos eventos, alterando o estado do sistema e possivelmente descobrindo ou encobrindo erros.

3.2 Depuração Baseada em Eventos

A execução do programa paralelo é vista como um ou várias seqüências de eventos como comunicação entre processos e eventos dentro de um processo. Uma característica importante deste tipo de depuração é que ele pode armazenar todo o histórico de eventos da execução do sistema, sendo que se um erro ocorrer, o desenvolvedor pode solicitar a execução do sistema de acordo com este histórico, podendo reproduzir o ambiente que causou o erro. A informação registrada para cada evento depende de como o histórico de eventos será manipulado. Basicamente o histórico é manipulado de três maneiras:

Browsing: utiliza o histórico para encontrar um determinado evento. É útil para se depurar cada evento separadamente. Este histórico não necessita de muitas informações sobre cada evento, apenas a sua ordem durante a execução.

Replay: o depurador usa o histórico de eventos para controlar uma nova execução do programa. Requer muitas informações para que os eventos de cada processo possam ser determinados.

Simulação: o histórico de eventos pode ser usado para simular um único processo, permitindo o uso de um depurador seqüencial sem necessitar executar todo o programa.

3.3 Fluxo de Controle

O fluxo de controle é uma representação de toda a comunicação e compartilhamento entre processos de um sistema. Ele pode ser representado em texto, diagramas, animações, ou outras formas, visando facilitar a visualização do que está ocorrendo ou já ocorreu no sistema para o desenvolvedor.

3.4 Análise Estática

A análise estática visa descobrir as falhas estruturais ao invés de identificar falhas funcionais, permitindo a eliminação de erros antes da execução de testes no sistema.

A técnica de análise estática é bastante utilizada em depuração paralela para detecção de certas classes de erros, principalmente quando o Efeito de Intrusão torna as técnicas anteriores inúteis (isto, porque esta técnica não altera o sistema, apenas o analisa antes de sua execução). Basicamente a análise estática é usada para detectar duas classes de erros: problemas de sincronização (deadlock e espera infinita) e problemas nas operações com dados (acesso simultâneo a variáveis compartilhadas). O problema desta abordagem é alto custo computacional da análise de programas paralelos, que em geral, é de complexidade exponencial.

3.5 Monitoração

Esta técnica permite monitorar o status da execução do software em tempo real. Ela é utilizada não mais na fase de desenvolvimento do software, mas é implementada juntamente com o software e distribuída com ele, permitindo que este software seja monitorado em sua execução em um ambiente real. Em geral, são monitoradas informações como status dos processos e nós, tráfego de informações e as transações do sistema. Este tipo de monitoração também é conhecido como geração de “logs” no sistema, informações que podem ser analisadas para se determinar o que causou uma falha.

Existem também técnicas e ferramentas para análise destas informações (já que geralmente são muitas informações, muitas vezes irrelevantes para a falha ocorrida).

4. Problemas com teste e depuração

As primeiras ferramentas desenvolvidas para a detecção de erros e a depuração de programas concorrentes acabaram não obtendo sucesso porque detectavam erros ou possíveis falhas onde não existiam falhas realmente. Assim, os programadores, ao utilizarem por algumas vezes e percebendo que a ferramenta mostrava muitos erros irrelevantes, preferiam deixar de utilizar estas ferramentas.

Isto aconteceu porque as ferramentas utilizavam Análise Estática para determinar possíveis erros no código, sendo que nem sempre ela conseguia determinar se este erro

poderia ou não ocorrer. Os algoritmos paralelos, em geral, podem ter partes onde não existe preocupação com o sincronismo e nem com proteção da memória, porque não existe dependência ou este controle pode ser feito pela idéia do algoritmo. Estas “falhas” eram todas apontadas pelo analisador, e por serem abundantes estas formas de controles implícitas no algoritmo, acabavam atrapalhando o desenvolvimento do software.

As ferramentas mais atuais, além de analisar estes possíveis erros, aplicam análises mais profundas para verificar se estas falhas podem ocasionar erros na execução do software.

5. Erros mais comuns e prevenção de erros

Uma das formas de se desenvolver ferramentas e técnicas para teste e depuração de erros em programas paralelos ou mesmo para se evitar erros é o de conhecer os erros mais comuns praticados pelos programadores.

Nesta seção mostraremos alguns dos erros mais comuns em programação concorrente. É bom observar que os exemplos aqui citados têm característica não determinística, ou seja, podem ou não funcionar, de acordo com a ordem dos eventos ocorridos. Por isso é bom que estes erros sejam evitados, facilitando bastante o processo de teste e depuração em sistemas paralelos.

- *Bibliotecas:*

O uso de bibliotecas em programas concorrentes é um ponto de introdução de falhas muito comum entre programadores inexperientes. Bibliotecas prontas geralmente possuem uma região de memória compartilhada e que pode ou não ser protegida. Bibliotecas que têm em sua implementação algum mecanismo de proteção para funcionar concorrentemente são chamadas threadsafe libraries. Bibliotecas que não são threadsafe não devem ser utilizadas em programas concorrentes (a não ser que seja usada por um único thread, sendo desta forma utilizada seqüencialmente).

Programadores experientes e empresas de desenvolvimento de software geralmente aplicam testes exaustivos sobre bibliotecas que serão utilizadas de forma concorrente, mesmo que sejam declaradas como threadsafe, visando diminuir as possibilidades de erros devido a código externo à aplicação sendo desenvolvida.

- *Assumindo ordem de execução:*

Programadores inexperientes geralmente esperam que os comandos geralmente sejam executados na ordem em que foram escritos. Esta visão é um resquício da programação estruturada a que todo programador é confrontado durante o período de aprendizado da computação. É preciso se desvencilhar da visão de ordem de execução dos processos.

No exemplo abaixo, quatro threads são criados e a cada um é passada a referência à variável “j” (observa-se que a referência é passada, ou seja, eles compartilham a variável). Pode-se observar que este programa apresenta um resultado não determinístico, pois o comando “printf” de um thread pode ser executado antes que o comando de incremento do thread principal sobre a variável “j” seja executado (ou vice-versa), resultando na impressão de números iguais e números não seqüenciais. Este é um exemplo de Race Condition, onde, dependendo da ordem de execução, o resultado final será diferente.

```
#include <stdio.h>
#include <pthread.h>
#define N 4

void *hello (void*);

int main() {
   int j;
   pthread_t tid[N];

   for (j = 0; j < N; j++)
pthread_create (&tid[j], NULL, hello, (void *)&j);

   for (j = 0; j < N; j++)
      pthread_join (tid[j], NULL);
}

void *hello (void *my_id) {
printf ("Hello World from thread %d\n", *(int *)my_id);
   return NULL;
}
```

- *Variáveis Mutex em escopo incorreto:*

Programadores inexperientes declaram variáveis Mutex em escopo incorreto, fazendo com que cada thread tenha um Mutex diferente, ou seja, a exclusão mútua nunca ocorre. No trecho de código abaixo, o Mutex é declarado dentro da função “icrement_counter”, que é utilizada por cada thread para incrementar a variável compartilhada “counter”. Pode-se observar claramente que o escopo do Mutex “lock” é dentro daquela função, dentro de um determinado thread, não afetando outros threads que podem estar executando esta mesma função.

```
int counter = 0;   /* shared counter */

void increment_counter (void) {
   pthread_mutex_t *lock;

   lock = (pthread_mutex_t *)
             malloc( sizeof(pthread_mutex_t) );
   pthread_mutex_init (lock, NULL);

   pthread_mutex_lock (lock);
      counter++;
   pthread_mutex_unlock (lock);
}
```

- *Perda de sinais:*

Quando variáveis de condição são utilizadas, é de total responsabilidade do programador sincronizar o envio do sinal de condição e o recebimento do mesmo. Observa-se que se o thread que está executando a função “signal” executar o comando “pthread_cond_signal” para enviar a mensagem de condição satisfeita e o thread que executa a função “wait” não estiver esperando o sinal através do comando “pthread_cond_wait” este sinal será perdido. Assim, o thread que execute “wait” esperará eternamente pelo sinal que se perdeu.

Este tipo de erro é muito comum quando se utiliza Pthreads, mas em algumas outras arquiteturas o sinal não é perdido, evitando este tipo de erro. Para previnir-se deste erro o

programador deve ter sempre certeza (através de sincronização) de que o sinal só seja enviado se estiver alguém esperando para recebê-lo.

```
#include <pthread.h>

void *signal (void*);
void *wait (void*);

int flag = 0;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cnd = PTHREAD_COND_INITIALIZER;

int main() {
   pthread_t tid1, tid2;

pthread_create (&tid1, NULL, signal, NULL);
   pthread_create (&tid2, NULL, wait, NULL);

   pthread_join (tid1, NULL);
   pthread_join (tid2, NULL);
}

void *signal (void *arg) {
   initialize_data();
   flag = 1;
   pthread_cond_signal (&cnd);
   return NULL;
}

void *wait (void *arg) {
   pthread_mutex_lock (&mtx);
      while (!flag)
         pthread_cond_wait (&cnd, &mtx);
      work_on_data();
   pthread_mutex_unlock (&mtx);
   return NULL;
}
```

Abaixo há uma seqüência típica de eventos onde ocorreria um deadlock no programa descrito acima (T1 → signal; T2 → wait):

T1 - initialize_data()

T2 - pthread_mutex_lock()

T2 - avalia condição: (!flag) é false, entra no "while"

T1 - flag = 1

T1 - envia sinal

T2 - espera pelo sinal (deadlock)

- *Locks pendurados (dangling locks):*

A operação lock em um Mutex deve sempre ser seguida por uma operação unlock. Um deadlock pode ocorrer sempre que um thread termina ou deixa a região crítica sem executar um unlock. Este tipo de erro é comum em programas seqüenciais que foram paralisados.

```
void *check_for_work (void *arg) {
   int data;

   while (1) {
      pthread_mutex_lock (&mtx);
         while (!new_data)
            pthread_cond_wait (&cnd, &mtx);
         data = buffer;
         if (data == 0) break;
         new_data = 0;
      pthread_mutex_unlock (&mtx);
      do_work (data);
   }
   return NULL;
}
```

No próximo exemplo, o programa pode ou não funcionar, dependendo do escalonamento entre os threads. Neste caso, um dos threads deixa a região crítica sem realizar um unlock, e pode causar um deadlock no outro thread se este ainda não foi executado.

```
#include <stdio.h>
#include <pthread.h>

void *race_to_lock (void*);

pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void *race_to_lock (void *id) {
   int my_id = *(int *)id

   pthread_mutex_lock (&mtx)
      printf ("Thread %d acquired mutex\n", my_id);
      if (my_id == 2) {
         printf ("Last thread in critical section\n");
         return NULL;
      }
   pthread_mutex_unlock (&mtx);

   return NULL;
}
```

```
int main() {
   int tid1 = 1, tid2 = 2;
   pthread_t t1, t2;

pthread_create (&t1, NULL, race_to_lock, (void *)&tid1);
pthread_create (&t2, NULL, race_to_lock, (void *)&tid2);

   pthread_exit (NULL);
}
```

- *Hierarquias de locks:*

A maioria dos exemplos aqui utiliza um único Mutex para o programa. Mas, em geral, os programadores utilizam mais de um Mutex, o que pode acarretar deadlocks por causa da hierarquia de locks em Mutexes. Este tipo de erro é muito comum em programas paralelos. O programa abaixo mostra um simples, porém esclarecedor, exemplo de bad-locking cycle (problema ocasionado por que a hierarquia de locks é incorreta). No caso, um thread faz um lock primeiro em um Mutex e depois em outro Mutex. O outro thread faz os locks na ordem inversa. O programa provavelmente irá rodar normalmente várias vezes, mas, se em algum momento os threads forem escalanados de forma que um thread aplique o lock em um Mutex e o outro comece a executar e faça um lock no outro Mutex, teremos então um mútuo deadlock. Veja o exemplo:

```
#include <stdio.h>
#include <pthread.h>

void *thread1 (void*);
void *thread2 (void*);

pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

int main() {
   pthread_t tid1, tid2;

   pthread_create (&tid1, NULL, thread1, NULL);
   pthread_create (&tid2, NULL, thread2, NULL);

   pthread_join (tid1, NULL);
   pthread_join (tid2, NULL);
}
```

```
void *thread1 (void *arg) {
   pthread_mutex_lock (&lock1);
      printf ("Thread 1 holding first lock\n");
      pthread_mutex_lock (&lock2);
printf ("Thread 1 holding second lock\n");
      pthread_mutex_unlock (&lock2);
   pthread_mutex_unlock (&lock1);
   return NULL;
}

void *thread2 (void *arg) {
   pthread_mutex_lock (&lock2);
      printf ("Thread 2 holding second lock\n");
      pthread_mutex_lock (&lock1);
printf ("Thread 2 holding first lock\n");
      pthread_mutex_unlock (&lock1);
   pthread_mutex_unlock (&lock2);
   return NULL;
}
```

- *Threads interrompidos:*

Um grande problema em sistemas paralelos ocorre quando um thread é terminado inesperadamente. Outro possível problema ocorre quando o sistema operacional não consegue alocar o número de threads exigido. Em ambos os casos, tanto um deadlock quanto um simples resultado errado pode ser obtido. No exemplo abaixo, um resultado errado pode ser obtido caso o sistema operacional não aloque algum dos threads requisitados.

```
#include <pthread.h>
#include <stdio.h>

#define N 32

void *worker (void*);

double delta, pi = 0.0;
int intervals = 100000;
pthread_mutex_t lck = PTHREAD_MUTEX_INITIALIZER;

int main() {
   int j, id[N];
   pthread_t tid[N];

   delta = 1.0 / (double)intervals;

   for (j = 0; j < N; j++) {
      id[j] = j + 1;
pthread_create (&tid[j], NULL, worker, (void *)&id[j]);
```

```
   }
   for (j = 0; j < N; j++)
      pthread_join (tid[j], NULL);

   printf ("Pi equals %f\n", pi * delta);
}

void *worker (void *id) {
   int j, start = *(int *)id;
   double x, partial_pi = 0.0;

   for (j = start; j <= intervals; j+=N) {
      x = ( (double)j - 0.5) * delta;
      partial_pi += 4.0 / (1.0 + (x * x));
   }
   pthread_mutex_lock (&lck);
      pi += partial_pi;
   pthread_mutex_unlock (&lck);
   return NULL;
}
```

Neste caso o sistema operacional é responsável apenas por responder que não conseguiu alocar mais threads, sendo que o programa deve tratar estas mensagens.

6. PVM e MPI

Apesar de existirem algumas ferramentas para se depurar um programa paralelo feito em PVM ou em MPI, usar a função “printf()” continua sendo o melhor método para se realizar tal tarefa.

Uma das maneiras de se depurar um programa em PVM é utilizando-se a “libpvm trace facility”, que permite, através da “trace mask”, percorrer cada função isoladamente e, através da “trace sink”, monitorar os dados que são enviados.

Tem-se abaixo alguns elementos do PVM com uma breve explicação sobre seu funcionamento e importância:

- PvmDebugMask: quando a depuração é ativada, PVM registrará informações detalhadas sobre as suas operações e sobre o seu progresso em stderr. O parâmetro val é o nível de depuração. O padrão é não imprimir qualquer informação de depuração;

- PvmAutoErr: quando um erro resulta de uma chamada a uma função da biblioteca PVM e PvmAutoErr está ativo (em um, padrão), uma mensagem de erro é automaticamente impressa em stderr. Definindo PvmAutoErr com zero desabilita a opção, enquanto, definindo ela com dois, faz com que a biblioteca termine a tarefa após a impressão da mensagem de erro;

O padrão MPI possui as mesmas dificuldades expostas pelo padrão PVM. Existem várias ferramentas que tentam ajudar no processo de depuração, no tópico seguinte serão comentadas e analisadas algumas destas ferramentas.

7. Ferramentas para PVM e MPI

Existem diversas ferramentas para se testar e depurar programas paralelos desenvolvidos utilizando-se os modelos PVM ou MPI. Estas ferramentas englobam operações como a análise estática (análise do código para detecção de erros comuns praticados pelos programadores), teste (auxílio à execução de testes e simulação de diversos ambientes não-determinísticos) e finalmente depuração (quando erros são detectados). Ainda existem diversas ferramentas (não listadas aqui) utilizadas para a monitoração de programas, tanto paralelos quanto seqüenciais, gerando informações em tempo real para depuração de possíveis problemas ou falhas no sistema.

Infelizmente, estas ferramentas ainda não apresentam o desempenho e a eficiência desejados, estando, em sua maioria, em fase puramente experimental. Existem algumas ferramentas comerciais que podem ajudar bastante no processo de teste e depuração, porém não é garantido que o seu objetivo seja alcançado, devido ao avanço que ainda é preciso se fazer nesta área.

As ferramentas que são mais utilizadas nas plataformas PVM e MPI podem ser vistas a seguir:

8. TotalView

Ferramenta produzida pela Etnus, que possui suporte para teste e depuração de programas PVM, MPI, OpenMP e threads nas linguagens Fortran, C e C++. Esta é uma ferramenta comercial (pode-se obter uma versão de avaliação no site www.etnus.com). Atualmente é tida como a melhor ferramenta para teste e depuração (de programas seqüenciais ou paralelos) existente para Linux e UNIX (possui suporte para quase todas as plataformas UNIX atuais). Permite análise do código e depuração (não permite realização de testes automáticos). Está atualmente na versão 6.2.

9. Distributed Debugging Tool (DDT)

Ferramenta produzida pela Streamline Computing, possuindo suporte para a maioria das implementações MPI, e as linguagens C, C++, Fortran 77 e Fortran 90. Permite fácil visualização do status de cada processo em execução e depuração linha a linha do código. Existem versões para Linux, Solaris e Alpha. É possível baixar uma versão de avaliação no site http://www.streamline-computing.com. A versão mais atual é a versão 1.2.

Outras ferramentas mais específicas sobre PVM são:

- Xmdb: utilização educacional.
- P2d2: depurador portátil desenvolvido pela NASA.
- AIMS: depurador também desenvolvido pela NASA.
- CXTRACE: ferramenta comercial baseada no AIMS.

10. Ferramentas Experimentais

Existem ferramentas experimentais desenvolvidas em projetos de pesquisa, que têm como objetivo estudar novos métodos principalmente na detecção de erros (e somente erros reais)

em softwares paralelos. São vários os estudos realizados na área, sendo alguns específicos de linguagens como Java, Fortran e C++. Estão em geral, ligadas mais à implementação de threads (depende de cada linguagem) e somente algumas tratam de MPI e PVM.

11. Conclusão

A realização de testes para se detectar a presença de erros em um programa paralelo (ou seqüencial) e o uso de ferramentas de depuração para encontra-los e corrigi-los, é uma tarefa fundamental para se garantir um software de qualidade.

Felizmente existe um grande número de métodos e técnicas já consagrados para teste e depuração de programas seqüenciais, o que não indica que não sejam necessárias pesquisas nesta área, pois ainda pode-se evoluir muito. Quanto aos programas paralelos, não existe o mesmo número de ferramentas de suporte, e as que existem não fornecem resultados tão precisos quanto as de programas seqüenciais. Sendo assim, fica bastante prejudicada a avaliação e manutenção de um programa paralelo.

Atualmente, as pesquisas voltadas para esta área têm como principais metas desenvolverem ferramentas eficientes para detecção e localização de erros e falhas nos códigos já prontos, mas também objetivam fazer com que o programador não cometa erros ou falhas, isto pode ser obtido através da aplicação de metodologias de programação.

Um dos grandes problemas ao se realizar testes e depuração de programas paralelos é a tendência natural de se tentar adaptar métodos consagrados da programação seqüencial para a programação paralela. Tal tentativa é normalmente frustrada pelo fato de que os principais problemas de cada um destes tipos de programação são completamente diferentes, ou seja, cada um dos tipos exige uma metodologia específica.

12. Bibliografia

[1] SANTANA, Regina Helena Carlucci; SANT'ANA, Tomás Dias; *ASTRAL – Ambiente de Simulação e Teste de pRogramas parALlos*, ICMC – USP e UNIFENAS.

[2] NETO, J. C. da C. *Teste Estrutural Baseado em Fluxo de Dados de Programas Concorrentes*, ICMC – USP.

[3] Etnus em http://www.etnus.com visitado em junho de 2003.

[4] IGLINSKI, Paul *Using a Template-Based Parallel Programming Environment to Eliminate Errors*, Department of Computing Science, University of Alberta, Canadá.

[5] PVM em http://www.csm.ornl.gov/pvm/pvm_home.html visitado em junho de 2003.

[6] MPI em http://www-unix.mcs.anl.gov/mpi/ visitado em junho de 2003.

[7] GABB, Henry Common Concurrent Programming Errors, Linux Magazine, março de 2002 em http://www.linux-mag.com/2002-03/concurrent_01.html visitado em junho de 2003.