Detecção de Breakpoints em Programas MPI

Marcelo L. N. Pinheiro, Lúcia M. A. Drummond
Pós-Grad. Ciência da Computação, UFF,
R. São Paulo, 24210-130, Niterói-RJ
e-mail:lucia@dcc.uff.br

Valmir C. Barbosa
Programa de Engenharia de Sistemas e Computação COPPE/UFRJ,
Caixa Postal 68511, 21945-970 Rio de Janeiro-RJ
e-mail: valmir@cos.ufrj.br

RESUMO

O processo de depuração de programas é vital para o desenvolvimento de aplicações eficientes. São poucas as ferramentas existentes para depuração de programas paralelos distribuídos.

Neste trabalho apresentamos uma ferramenta que permite a detecção de *breakpoints* em programas baseados no padrão MPI. A ferramenta consiste essencialmente em analisar o programa fonte desenvolvido pelo usuário, acrescentando e substituindo código necessário para a posterior execução do mesmo, visando identificar a ocorrência do *breakpoint* que se tem interesse verificar. São implementados algoritmos para os seguintes tipos de *breakpoints*: incondicionais, baseados em predicados conjuntivos e disjuntivos.

A ferramenta é avaliada através de sua utilização na detecção de *breakpoints* em uma aplicação paralela baseada em algoritmos genéticos para solução do Travelling Purchaser Problem.

ABSTRACT

The process of debugging is vital to the development of efficient applications. In distributed environments there are few tools for parallel program debugging.

In this work we present a tool that makes it possible to detect breakpoints in programs based on MPI standard. The tool works by analyzing the users source code and adding or changing code as needed in order to perform the detection of the breakpoint the user is interested in. We present the implementation of four algorithms related to the following types of breakpoints: unconditional breakpoints, breakpoints based on disjunctives and conjunctive predicates.

The resulting tool is tested by applying it to detect breakpoints in an application based on parallel genetic algorithms to solve the travelling purchaser problem.

1 Introdução

A depuração consiste na análise e localização de erros. Enquanto em ambientes de programação seqüencial, existem disponíveis ferramentas que possibilitam ao programador a depuração de seu código, em programas paralelos distribuídos não há a disponibilidade de ferramentas eficientes com este propósito (veja [6]).

O problema de depuração de programas paralelos pode ser abordado a partir de duas perspectivas complementares. A primeira perspectiva trata da repetição determinística de programas paralelos distribuídos, cujo fluxo de eventos pode variar de execução para execução, impedindo a utilização da técnica de depuração cíclica, geralmente empregada na depuração de programas seqüencias [5]. A segunda perspectiva refere-se à detecção de *breakpoints*, que são expressos por uma combinação de condições espalhadas entre os diversos processos que constituem o programa paralelo distribuído [2].

Neste trabalho, apresentamos uma ferramenta que realiza a detecção de *breakpoints* em programas paralelos distribuídos que utilizam o padrão MPI [7]. O funcionamento da ferramenta consiste em analisar o programa fonte desenvolvido pelo usuário, substituindo e acrescentando código necessário para a posterior execução do mesmo, visando identificar a ocorrência do *breakpoint* desejado. Na estratégia adotada, as chamadas efetuadas a procedimentos MPI são substituídas no programa original do usuário por rotinas equivalentes, porém com acréscimo de função.

A ferramenta é baseada nos algoritmos de detecção de predicados globais apresentados por Drummond e Barbosa [2]. Este algoritmos são os primeiros algoritmos distribuídos eficientes que realizam a detecção do primeiro estado global que satisfaz um dos seguintes tipos de *brekpoints*: incondicionais, baseados em predicados conjuntivos e predicados disjuntivos.

A implementação realizada exigiu também que se adotasse uma estratégia de terminação da execução da ferramenta. Utilizamos, para isso, o algoritmo de terminação de aplicações distribuídas proposto por Shing-Tsann [10].

Testes preliminares da ferramenta proposta neste trabalho foram realizados utilizando-se uma aplicação baseada em um algoritmo genético paralelo para a solução do Travelling Purchaser Problem (veja [8]).

Nas seções seguintes apresentamos os algoritmos de detecção de predicados globais (Seção 2), a abordagem para resolução do problema, detalhando a estratégia de implementação dos algoritmos de detecção (Seção 3), o algoritmo de terminação (na Seção 4) e, finalmente, os testes e conclusão deste trabalho (seções 5 e 6).

2 Detecção de Breakpoints

Um aspecto fundamental na depuração de programas paralelos distribuídos consiste no estabelecimento de *breakpoints* que envolvem mais de um processo. A detecção de um *breakpoint* expresso por um predicado global, ou seja, uma condição que envolve mais de um processo, é um problema devido ao fato de os processos não terem acesso a um relógio global.

O trabalho apresentado por Drummond e Barbosa [2] apresenta quatro algoritmos distribuídos para a detecção de estados globais que satisfazem *breakpoints* incondicionais, disjuntivos, conjuntivos estáveis e conjuntivos genéricos. No mesmo

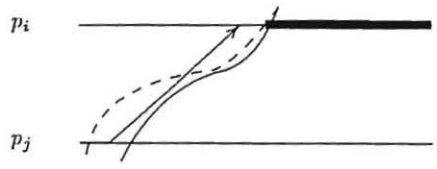

Figura 1: Detecção de predicados disjuntivos

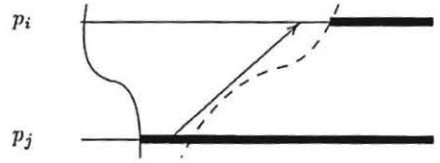

Figura 2: Detecção do estado global mais adiantado que satisfaz um predicado disjuntivo

trabalho ainda é apresentado um algoritmo de propagação de informação global, cujo objetivo é simplificar a compreensão dos demais algoritmos apresentados.

Vejamos, inicialmente, as principais características dos algoritmos para detecção de predicados disjuntivos e para propagação de informação global, que são aplicadas no projeto dos demais algoritmos.

No algoritmo de detecção de predicados disjuntivos, onde pelo menos uma das condições locais precisa ser satisfeita para que o predicado seja também satisfeito, são usadas apenas mensagens da própria aplicação, chamadas aqui de mensagens de computação, com alguns campos anexados. Um dos campos é um vetor de tempos, que neste trabalho representa o estado global [1]. Na Figura 1, onde os processos são representados por eixos de tempos locais, as setas representam mensagens de computação, as barras mais grossas representam os períodos durante os quais as condições locais são satisfeitas e o corte um estado global, p_j envia a mensagem de computação para p_i com um vetor de tempos. O processo p_i, ao recebê-la, atualiza seu próprio vetor de tempos (com os maiores valores entre os dois vetores) e, ao detectar a satisfação do predicado disjuntivo, determina o estado global que o satisfaz (o indicado pela linha cheia). A linha tracejada representa um estado global inconsistente.

Outro campo anexado à mensagem de computação, neste algoritmo, é o *status bit*, que determina se o predicado já foi detectado. Na Figura 2, p_j envia para p_i uma mensagem de computação com o *status bit* indicando que o predicado já foi satisfeito (estado representado pela linha cheia), impedindo que p_i detecte outro estado (o indicado pela linha tracejada). Este *bit* garante a detecção do estado global mais adiantado.

O algoritmo de propagação de informação global detecta o predicado conjuntivo em uma aplicação que não envia mensagens de computação, conforme Figura 3. Para isso, é executado um *broadcast* por enchente dos vetores de condições e tempos, onde as mensagens geradas são chamadas de mensagens de *broadcast*, cada vez que uma condição local é satisfeita. O vetor de condições é um vetor de variáveis lógicas, que

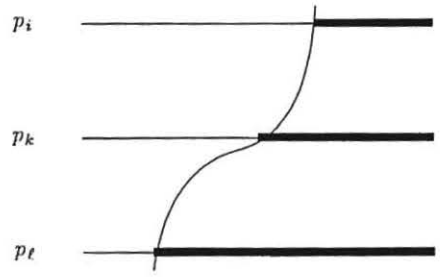

Figura 3: Propagação de informação global

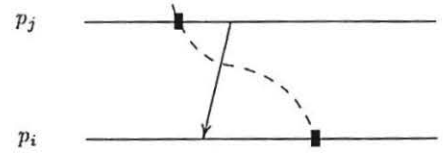

Figura 4: Erro na colocação dos *breakpoints* incondicionais locais

representa os estados das condições locais. Ao serem recebidas, estas mensagens atualizam os vetores dos processos destinos, que testam se o predicado conjuntivo foi satisfeito. Isto ocorre quando todos os elementos do vetor de condições tiverem seus valores verdadeiros. O vetor de tempos indica o estado global em que esta detecção ocorre.

O algoritmo de detecção de *breakpoints* incondicionais, definidos por pontos distribuídos pelos processos, além de empregar os dois tipos de mensagens (computação e *broadcast*), trata do problema de erros. Erros acontecem quando os pontos (*breakpoints* incondicionais locais) estabelecidos pelo usuário não determinam um estado global consistente, veja Figura 4.

Outra questão se refere à detecção do estado global mais adiantado que satisfaz o *breakpoint* incondicional, uma vez que pode acontecer que alguns processos não tenham *breakpoint* associado. Para tratar desta questão introduzimos vetores de tempos com visões alternativas do estado global nestes processos. As mensagens de computação atualizam estes vetores de tempos alternativos enquanto as mensagens de *broadcast* atualizam os vetores de tempos e os vetores de tempos alternativos. Na Figura 5, a linha cheia representa o estado global a ser detectado e a linha tracejada uma visão alternativa.

A detecção de predicados conjuntivos estáveis, onde os predicados locais uma vez verdadeiros permanecem verdadeiros até o final da execução, pode ser considerada uma simplificação do caso anterior, uma vez que não é mais necessária a detecção de erros. Assim, os algoritmos empregam basicamente a mesma técnica.

Para detecção de predicados conjuntivos genéricos não somente utilizamos vetores de tempos alternativos, mas também usamos vetores de condições alternativos. O *broadcast* neste caso deve conter a identificação da origem do processo que o iniciou. Na Figura 6, temos dois estados globais que satisfazem o predicado. Desejamos detectar o mais adiantado. Cada vez que a condição local se torna verdadeira é ini-

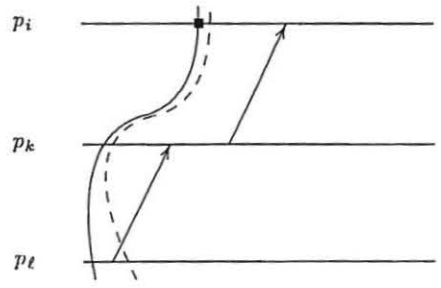

Figura 5: Estado global mais adiantado que satisfaz o *breakpoint* incondicional

ciado um *broadcast*. Neste caso, o segundo *broadcast* do processo p_i, por exemplo, é desconsiderado, pois os outros processos já mantêm a informação de que a condição local de p_i é verdadeira.

Na Figura 7, a linha cheia representa o estado a ser detectado e o segundo *broadcast* de p_i deve ser considerado, pois enquanto a condição local de p_i era falsa, este enviou uma mensagem para p_j, que registrou esta informação nos seus vetores. Outro aspecto importante neste algoritmo se refere ao fato de que ele é baseado em canais FIFO. Na Figura 8, p_i recebe de p_j uma mensagem de *broadcast* depois de uma mensagem de computação, e mais tarde uma outra mensagem de *broadcast*. Caso a última mensagem de *broadcast* chegasse em p_i antes da mensagem de computação, esta seria desconsiderada e o estado global, representado pela linha cheia, não seria detectado.

3 Descrição da Ferramenta

A ferramenta implementada pode ser dividida em duas partes: análise e detecção.

Na fase de análise, o programa aplicativo, no qual se deseja efetuar a detecção de um predicado global qualquer, é processado por um programa analisador que efetua a substituição dos comandos MPI originais pelos comandos equivalentes, porém com acréscimo de funcionalidade, desenvolvidos em nossa aplicação. Durante esta fase é gerado um arquivo temporário contendo o programa aplicativo original modificado. Na fase de detecção, o programa resultante da fase de análise é executado, sendo criada uma quantidade de processos de acordo com o especificado pelo usuário na fase anterior. Ainda durante a execução, são gravados arquivos de *log*, onde podem ser registrados diversos eventos pertinentes à execução de cada processo, tais como, envio e recebimento de mensagens e o valor do vetor de tempos no momento em que um dos processos detecta a ocorrência do predicado global.

Na abordagem utilizada, a aplicação distribuída é executada atrelando-se a cada um dos seus processos regulares, um processo depurador, que tem por missão interceptar todas as mensagens oriundas ou destinadas ao processo a que está associado, acrescentando ou retirando destas mensagens, informações necessárias à detecção do *breakpoint* desejado.

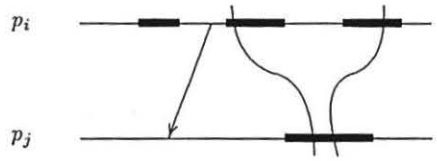

Figura 6: Dois estados globais onde o predicado conjuntivo é satisfeito

3.1 Análise

Durante a análise do programa original, além da substituição de comandos MPI, conforme Figura 9, são incluídas rotinas de inicialização e terminação do processamento. O código gerado após esta etapa é executado de forma controlada visando a detecção do predicado global desejado.

Nesta etapa são solicitadas ao usuário as seguintes informações: nome do programa fonte, o tipo de *breakpoint* a ser detectado, a quantidade de processos executados, a identificação dos processos que participam da detecção do predicado global e a descrição dos predicados locais que deverão ser verificados nos processos participantes.

O programa do usuário a seguir realiza o envio de uma mensagem do processo de *rank* 0 para o processo de *rank* 1.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
void main(argc, argv)
int argc;
char **argv;
{
int varia, varib;
int myrank;
int meu_buffer[1000];
int rec_buffer[1000];
int i;
MPI_Status meu_status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
printf("Processo %i executando!\n", dbg_myrank);
varia=5;
if (myrank==0) {
  for(i=1; i<5; i++){
    meu_buffer[i-1]=i*2;
  }
  MPI_Send(&meu_buffer,4,MPI_INT, 1, 45, MPI_COMM_WORLD);
};
if (myrank==1){
  MPI_Recv(rec_buffer,4,MPI_INT,0,45,MPI_COMM_WORLD,&meu_status);
  for (i=0;i<4;i++) {
    printf("i=%i\n",rec_buffer[i]);
  };
};
varia=5;
varib=6;
MPI_Finalize();
```

```
printf("Processo%i encerrou sua execucao\n",dbg_myrank);
};
```

O programa a seguir apresenta o programa anterior após a fase de análise, onde o predicado local a ser detectado no processo de *rank* 0 é a variável "varia" assumir o valor 5 e o predicado local a ser detectado no preocesso de *rank* 1 é a variável "varib" assumir o valor 6.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include "dbg.h"
void main(argc, argv)
int argc;
char **argv;
{
int varia, varib;
int myrank;
int meu_buffer[1000];
int rec_buffer[1000];
int i;
MPI_Status meu_status;
MPI_Init(&argc,&argv);
#include "depurador.h"
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
printf("Processo %i executando!\n", dbg_myrank);
varia=5;
switch(dbg_myrank){
case 0: if (varia==5) dbg_local_true();
  if (!(varia==5) dbg_local_false();break;
case 1: if (varib==6) dbg_local_true();
  if (!(varib==6) dbg_local_false();break;
default: break;
};
if (myrank==0) {
  for(i=1; i<5; i++){
    meu_buffer[ i-1 ] =i*2;
  }
  dbg_send(&meu_buffer,4,MPI_INT, 1, 45, MPI_COMM_WORLD);
};
if (myrank==1) {
  dbg_recv(rec_buffer,4,MPI_INT,0,45,MPI_COMM_WORLD,&meu_st

  for (i=0;i<4;i++){
    printf("i=%i\n",rec_buffer[i]);
  };
};
varia=5;'
switch(dbg_myrank) {
case 0: if (varia==5) dbg_local_true();
  if (!(varia==5) dbg_local_false();break;
case 1: if (varib==6) dbg_local_true();
  if (!(varib==6) dbg_local_false();break;
default: break;
};
varib=6;
switch(dbg_myrank){
case 0: if (varia==5) dbg_local_true();
```

```
  if (!(varia==5) dbg_local_false();break;
case 1: if (varib==6) dbg_local_true();
  if (!(varib==6) dbg_local_false();break;
default: break;
};
dbg_finaliza();
MPI_Finalize();
printf("Processo%i encerrou sua execucao\n",dbg_myrank);
};
```

Além das substituições de comandos MPI, o analisador inclui as bibliotecas dbg.h e depurador.h no programa do usuário, que têm como finalidade acrescentar ao código do usuário os procedimentos e estruturas de dados necessários para o funcionamento da rotina de detecção de predicados globais e incluir o código necessário aos processos depuradores para o gerenciamento de mensagens durante o processamento da aplicação. Ainda é inserido no programa original uma série de comandos, cujo objetivo é identificar se o *breakpoint* local foi atingido, sempre que se verificar a possibilidade das variáveis envolvidas na composição do predicado local sofrerem algum tipo de modificação acarretando alteração no valor lógico do predicado local.

Os comandos que substituem os originais do MPI, permitem que os processos depuradores interceptem as mensagens trocadas entre os processos regulares, acrescentando e retirando informações tais como vetores de tempos e condições.

O comando dbg_finaliza é responsável por indicar ao processo depurador que o processo regular, a ele associado, encerrou seu processamento, iniciando, pelo depurador, o processo de terminação distribuída, descrito na Seção 4.

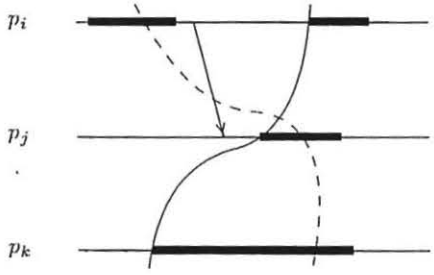

Figura 7: Detecção de predicados conjuntivos genéricos

3.2 Detecção

Como já foi dito, os processos depuradores interceptam as mensagens trocadas entre os processos regulares acrescentando ou retirando informação. Cada mensagem enviada pelos processos MPI é caracterizada por um *tag*.

Os procedimentos desenvolvidos em nossa ferramenta estão em geral associados a mensagens geradas em decorrência de eventos que ocorrem nos processos que participam da aplicação do usuário. Desta forma, quando o predicado local associado a um processo regular se torna verdadeiro, é enviada uma mensagem do processo regular para o processo depurador correspondente com um *tag* específico, que indicará que sejam realizadas atualizações nos seus vetores de tempos e condições e que

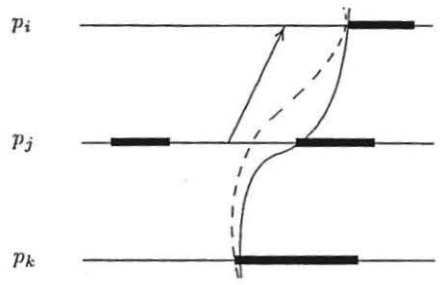

Figura 8: Predicado conjuntivo genérico e canais FIFO

seja iniciado um *broadcast*, dependendo do tipo de *breakpoint* que se deseja detectar. Outros tipos de *tags* são usados para controlar o fluxo de mensagens necessários para executar os comandos MPI originais acrescidos da funcionalidade exigida pelo depurador.

Vejamos, por exemplo, o que ocorre quando uma mensagem é enviada do processo p_i para o processo p_j através do dbg_send, que substitui o MPI_Send, e é recebida pelo dbg_recv, que substitui o comando MPI_Recv. O procedimento dbg_send monta um pacote para transmissão, contendo as seguintes informações: o conteúdo da mensagem original, conforme especificado no comando MPI_Send, a identificação (*rank*) do processo que originou a mensagem, o tamanho da mensagem original, o *rank* do destinatário, o *tag* original usado no comando MPI_Send que foi substituído. Em seguida, o procedimento dbg_send se utiliza do comando MPI_Send para efetuar o envio deste pacote para o processo depurador associado ao processo regular em questão. A mensagem acima é enviada com o *tag* DBG_TAG_COMPUTACAO. O processo depurador, ao receber uma mensagem com este *tag*, executa um procedimento que acrescenta ao pacote recebido os vetores de tempo e condição rotulando-o de DBG_TAG_QQ e o envia, utilizando o comando MPI_Send para o processo depurador associado ao processo destinatário da mensagem original. Um processo depurador, ao receber uma mensagem com este *tag*, retira do pacote recebido as informações referentes a vetores de tempo e condição e envia uma mensagem ao processo regular, a ele associado, informando que existe uma mensagem disponível para ser recebida cujo *tag* é uma combinação da identificação do processo origem e o *tag* da mensagem original. Esta estratégia é empregada pois o procedimento de recebimento de recepção empregado originalmente no programa do usuário também é substituído por uma das rotinas de nossa ferramenta, e com isso, a origem da mensagem sempre será o processo depurador associado ao processo regular destinatário e não mais o processo que enviou a mensagem original. Ao executar o procedimento dbg_recv, é recebida a mensagem cujo *tag* é uma combinação do *tag* orignal e a identificação do processo origem da mensagem.

Os demais comandos relacionados na Figura 9 são implementados de forma análoga. É importante ressaltar que o fato de nossa ferramenta introduzir processos depuradores no trajeto das mensagens entre os processos regulares, impossibilita a utilização de instruções que assumem canais de capacidade zero, o que poderá acarretar efeitos colaterais no sincronismo desejado pelos usuários de comandos bloqueantes.

Comando MPI	Comando usado pela ferramenta
MPI_Send	dbg_send
MPI_Bsend	dbg_bsend
MPI_Rsend	dbg_rsend
MPI_Ssend	dbg_ssend
MPI_Isend	dbg_isend
MPI_Ibsend	dbg_ibsend
MPI_Issend	dbg_issend
MPI_Irsend	dbg_irsend
MPI_Recv	dbg_recv
MPI_Irecv	dbg_irecv
MPI_Broadcast	dbg_broadcastmp
MPI_Gather	dbg_gather
MPI_Scatter	dbg_scatter

Figura 9: Tabela de substituição de comandos

As mensagens de *broadcast* empregadas nos algoritmos de detecção são rotuladas com o *tag* DBG_TAG_BROADCAST. Ao serem recebidas, poderá ocorrer a atualização dos vetores de tempos e condições e, relaiza-se, se necessário, a execução do procedimento que dará continuidade ao *broadcast*.

Por fim, devemos mencionar que o último procedimento executado pelos processos regulares envia uma mensagem com o *tag* DBG_TAG_FIM para o processo depurador associado, indicando que o processamento de sua tarefa terminou. Ao receber uma mensagem deste tipo o processo depurador irá executar um algoritmo de terminação distribuída, responsável pelo término do conjunto de processos depuradores utilizados em nossa ferramenta.

Diversos outros tipos de *tag* são empregados em nossa ferramenta, e estão detalhados em [9].

4 Algoritmo de Terminação

O problema de terminação de aplicações distribuídas tem atraído considerável atenção [3] [4] [10]. Em nosso trabalho para finalizarmos os processos depuradores empregamos o algoritmo proposto por Shing-Tsaan [10]. O algoritmo considera a existência de canais bidirecionais entre cada par de processos e se baseia no trabalho de Chandy e Lamport [1] para gravação de estados globais em computações distribuídas com canais de comunicação FIFO.

O algoritmo considera que um processo está em um dos seguintes estados: suspenso ou ativo. Um processo ativo pode ter seu estado alterado para suspenso a qualquer momento. Apenas os processos ativos podem enviar mensagens e processos suspensos podem ser ativados através do recebimento de uma mensagem. A computação distribuída pode ser considerada como encerrada se todos os processos estão no estado suspenso e não há nenhuma mensagem em trânsito. Em nossa implementação procedemos de forma que, quando um dos processos regulares encerra

seu processemanto, este envia uma mensagem para o processo depurador a ele associado, que por sua vez entra no estado suspenso iniciando a gravação do "retrato" do sistema. O retrato obtido pelo processo indica se este pode ou não terminar. Ao final da gravação do retrato, os processos depuradores enviam mensagens para um processo depurador responsável por manter um registro dos estados dos processos depuradores. Através deste registro, este processo é capaz de identificar se todos os processos podem encerrar seus processamentos, quando então seria enviada uma mensagem de encerramento para cada um dos processos depuradores.

5 Testes Preliminares

Com o objetivo de avaliar a ferramenta implementada, realizamos testes com uma aplicação baseada em algoritmos genéticos paralelos para resolver o Travelling Purchaser Problem (TPP) [8].

Para descrever o TPP, necessitamos dos seguintes dados:

- Um conjunto de n pontos de venda $N = \{1, 2, \ldots, n\}$ e uma origem $s = \{0\}$
- Um conjunto de m produtos $K = \{1, 2, \ldots, m\}$ que podem ser vendidos nos n pontos de venda
- Uma matriz $D = (d_{kj})$ tal que $k \in K$, $j \in N - \{0\}$, onde d_{kj} é o custo do item k no ponto de venda j
- Uma matriz $C = (c_{ij})$ tal que $i, j \in N$, onde c_{ij} é o custo da viagem de i até j.

Assume-se que:

- Cada um dos itens está disponível para venda em ao menos um dos pontos de venda $j \in N - \{0\}$
- Nenhum item pode ser comprado em $s = \{0\}$
- O comprador pode passar por qualquer um dos pontos de venda sem necessariamente adquirir quaisquer itens disponíveis
- O comprador pode adquirir tantos itens quantos estejam disponíveis em um determinado ponto de venda por ele visitado.

Nós definimos um grafo completo direcionado $G = (N, A)$ sem *loops*, onde N é o conjunto de vendas já definidos e cada arco $(i, j) \in A$ representa uma alternativa de caminho de i para j. Uma matriz (d_{ik}) tal que $k \in K$ é associada com cada vértice (ponto de venda) $i \in N$, isto é, o custo do item k no vértice i. O custo da viagem de i para j é dado por c_{ij}, tal que $(i, j) \in A$.

O objetivo do TPP é obter um ciclo direcionado incluindo a origem s e passando através de um subconjunto $J \subseteq N$ tal que o custo total de deslocamento e aquisição dos produtos seja minimizado.

O programa teste utilizado é uma aplicação genética paralela para solução do TPP. De forma análoga ao processo de reprodução genética, cada solução viável do

problema é representada como um cromossomo e cada cromossomo é avaliado com relação às restrições que se aplicam sobre o problema. Os melhores cromossomos, isto é, as melhores soluções obtidas até o momento, são recombinados visando a obtenção de novos cromossomos que possam vir a representar melhores soluções para o problema. A técnica utilizada na implementação do programa consiste em criar diversos processos, que tentam paralelamente solucionar o TPP. Cada um dos processos trabalha sobre uma subpopulação inicial de cromossomos (soluções). A cada geração em que um cromossomo, mais próximo da solução ótima, é obtido, realiza-se a migração do mesmo para os demais processos. O cromossomo migrante é utilizado, pelo processo que o recebeu, para substituir o pior cromossomo da subpopulação local.

O programa genético paralelo para solução do TPP, por nós desenvolvido foi executado sobre 12 produtos, 30 pontos de venda e 500 cromossomos na população inicial. Os testes foram realizados com 2, 4, 6 e 8 processadores, utilizados de forma exclusiva no SP/2. Não foram percebidas alterações do resultado do programa original que pudessem ser consideradas como um efeito colateral devido ao uso de nossa ferramenta. Efetuamos testes para cada um dos algoritmos de detecção implementados. No caso de *breakpoints* incondicionais estabelecemos como ponto de parada em cada processo que compõe a aplicação o ponto imediatamente anterior ao início do ciclo de geração das subpopulações. Considerando a detecção de *breakpoints* disjuntivos, estabelecemos que o predicado global seria satisfeito quando pelo menos um dos processos conseguisse obter uma solução local melhor do que as obtidas nas iterações anteriores. Para *breakpoints* conjuntivos estáveis consideramos a detecção de um estado global em que todos os processos atingissem um certo grau considerado como ótimo. Este grau é o resultado de uma função de avaliação dos cromossomos de uma população utilizada pelo programa genético e reflete as restrições que se aplicam às possíveis soluções, onde graus tendentes a zero representam soluções mais próximas do ideal. O algoritmo de detecção do predicado conjuntivo genérico foi testado, estabelecendo-se como predicado global o estado em que a solução migratória recebida fosse pior do que a pior solução obtida no ciclo reprodutivo anterior em todos processos.

O gráfico a seguir, apresenta os tempos de execução em segundos da aplicação original e da mesma com o depurador, para cada um dos casos descritos acima (da esquerda para direita: aplicação original, *breakpoints* incondicional, disjuntivo, conjuntivo estável e conjuntivo genérico), utilizando 2, 4, 6 e 8 processadores. O acréscimo de tempo (na média de 430%) deve-se principalmente às mensagens adicionais trocadas entre os processos depuradores, ao aumento do número de instruções devido ao código inserido por nossa ferramenta e à constante gravação de mensagens no arquivo de *log*.

6 Conclusão

Neste trabalho, implementamos uma ferramenta que permite a detecção de predicados globais em programas paralelos distribuídos desenvolvidos no padrão MPI. Para efeito de testes foi utilizado um programa baseado em uma versão simplificada do algoritmo genético paralelo para a solução do Travelling Purchaser Problem.

A principal contribuição deste trabalho consiste na implementação de algoritmos

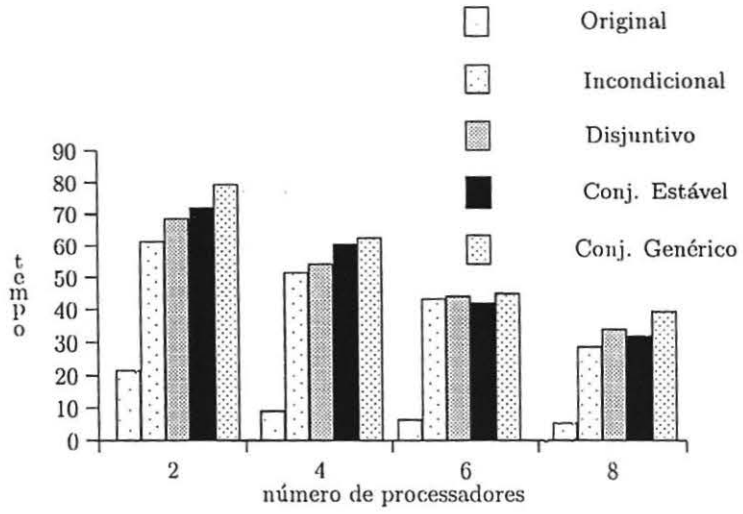

Figura 10: Resultados preliminares

de detecção de *breakpoints* que poderão ser utilizados em um sistema de depuração de aplicações desenvolvidas com base no padrão MPI.

Como continuação do trabalho, estamos desenvolvendo uma ferramenta que possibilita a re-execução determinística de programas MPI. Desta forma, poderemos empregar a técnica de depuração cíclica, associada aos procedimentos de detecção de *breakpoints*. Testes mais detalhados deverão ser realizados para avaliar a ferramenta nesta nova etapa do trabalho.

Bibliografia

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

[2] Drummond, L. M. A., and Barbosa, V. C. Distributed breakpoint detection in message-passing programs. *Journal of Parallel and Distributed Computing 39* (1996), 153–167.

[3] Dijkstra, E. W., Scholten, C. S. Termination detection for distributed computation. *Inform. Processing Letter 11* (1980), 1–4.

[4] Francez, N. Distributed termination. *ACM Trans. on Programming Language and Systems 2* (1980), 42–55.

[5] LeBlanc, T., Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36* (1985), 471–482.

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

[7] Snir, M., Otto, S. W., Huss, S., Walker, D., Dongarra, J. MPI The Complete Reference, *THE MIT PRESS* (1996).

[8] Ochi, L. S, Drummond, L. M. A., Figueiredo, R. M. V., Design and implementation of a parallel genetic algorithm for the travelling purchaser problem, *ACM Symposium on Applied Computing* (1997), 215–221.

[9] Pinheiro, M. L. N. Detecção de predicados globais em programas MPI, *Tese de Mestrado*, Universidade Federal Fluminense (1997).

[10] Shing-Tsaan, H. Termination detection by using distributed snapshots, *Inform. Processing Letters 32* (1980),