Alexandre Ceolin Hausen

ValiMPI: Uma Ferramenta de Teste Estrutural para Programas Paralelos em Ambiente de Passagem de Mensagem

Curitiba

2005

Alexandre Ceolin Hausen

ValiMPI: Uma Ferramenta de Teste Estrutural para Programas Paralelos em Ambiente de Passagem de Mensagem

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná

Orientadora: Prof.ª Dr.ª Silvia Regina Vergilio

Universidade Federal do Paraná

Curitiba

2005

Dissertação de Mestrado sob o título *"ValiMPI: Uma Ferramenta de Teste Estrutural para Programas Paralelos em Ambiente de Passagem de Mensagem"*, defendida por Alexandre Ceolin Hausen e aprovada em 27 de setembro de 2005, em Curitiba, Estado do Paraná, pela banca examinadora constituída pelos doutores:

Prof.ª Dr.ª Silvia Regina Vergilio
Orientadora

Prof. Dr. Sérgio Scheer
Universidade Federal do Paraná

Prof. Dr. Elias Procópio Duarte Jr.
Universidade Federal do Paraná

Aos meus pais João Rogério e Neusa,
à minha namorada Bárbara e
à minha irmã Caroline

Agradecimentos

Agradeço inicialmente a minha orientadora, Prof.ª Dr.ª Silvia Regina Vergilio, pelo apoio, motivação e críticas construtivas que tornaram esse trabalho uma realidade.

Aos Professores do grupo de pesquisa ValiPVM, Prof.ª Dr.ª Simone do Roccio Senger de Souza e Prof. Dr. Paulo Sérgio Lopes de Souza, pelas sugestões e hospitalidade.

Aos alunos de iniciação científica da UFPR, Eric e Augusto, e aos alunos de iniciação científica da UEPG, Alexandre, Thiago e Bruno, pela prontidão nas respostas às minhas solicitações e pelo esforço empregado na construção das ferramentas.

Aos Colegas de laboratório, Marcos, Araújo e Rubens, pelas sugestões e auxílio no uso do LaTeX e também pelas conversas descontraídas que tivemos.

À minha irmã Caroline e ao meu primo Lucas, por apontarem erros de português (essa língua melindrosa) nas versões preliminares da dissertação.

A todos aqueles que, de alguma forma, me apoiaram e incentivaram no decorrer desse trabalho.

Resumo

Aplicações que demandam processamento intenso e exploram o paralelismo para reduzir o tempo de computação são usadas nos mais variados domínios. Para isso, existem vários paradigmas de programação paralela, dos quais o paradigma de passagem de mensagem é um dos mais utilizados. Dentre os ambientes de passagem de mensagem destaca-se o MPI (*Message Passing Interface*), um padrão para o desenvolvimento de aplicações paralelas. Uma falha nessas aplicações pode representar altos custos, portanto, a realização de atividades de garantia de qualidade, como o teste de software, é fundamental. Como é inviável testar o programa para todas as entradas possíveis, o usuário deve usar critérios para guiar a escolha dos casos de teste com maior probabilidade de revelar erros. Os critérios estruturais destacam-se pela cobertura do código. Como os critérios de teste para programas seqüenciais não são adequados para programas paralelos cresceu a motivação na pesquisa de novos critérios para a programação paralela. O projeto ValiPVM introduziu critérios específicos para testar programas paralelos por passagem de mensagem, entretanto a aplicação efetiva desses critérios requer o uso de uma ferramenta de teste. Esta dissertação descreve aspectos da implementação da ValiMPI, uma ferramenta de teste para programas paralelos na linguagem C e MPI.

Abstract

Computationally intensive applications which use parallelism to reduce computing time are widely used. There are several parallel programming paradigms, message passing is one of the most popular. Among the message passing environments, MPI (Message Passing Interface) emerged as a standard for developing parallel applications. A failure in these application may represent high costs, therefore software testing is an essential activity for sotware quality assurance. However, testing all possible input data is not feasible, so the user must use some criteria in order to choose test cases most likely to reveal errors. Structural testing criteria offer code coverage measures that allow the evaluation of a test set. Because of this, traditional criteria have been extended to the context of parallel programing. This project, named ValiPVM, introduced some specific criteria for testing message passing systems, however, in order to use use of these criteria effectively a testing tool is required. This work describes ValiMPI, a tool that implements the proposed criteria for testing parallel programs in C an MPI.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Listagens

1 Introdução

Neste capítulo serão apresentados o contexto no qual este trabalho está inserido, a motivação e relevância para realizá-lo, os objetivos visados e a organização desta dissertação.

1.1 Contexto

Aplicações que demandam grande poder computacional e exploram o paralelismo para reduzir o tempo de computação são usadas nos mais variados domínios. O uso de paralelismo é crucial na resolução de problemas que demandam alta capacidade de processamento, como em aplicações de simulação, processamento de imagens, inteligência artificial, bioinformática, entre outras [51, 60]. A recente popularização dos *clusters* de computadores, a padronização de bibliotecas para programação paralela multiplataforma e o surgimento de novas aplicações que demandam maior poder computacional favorecem o desenvolvimento de programas paralelos [46].

Existem vários paradigmas para a construção de programas paralelos, entre eles: tarefas e canais, paralelismo de dados, memória compartilhada e passagem de mensagem [16]. O paradigma de passagem de mensagem pode ser implementado tanto em multicomputadores quanto em multiprocessadores, sendo um dos paradigmas mais usados. Dentre os ambientes de passagem de mensagem destacam-se o PVM (*Parallel Virtual Machine*) [19] e o MPI (*Message Passing Interface*) [36, 37]. O MPI surgiu como uma tentativa de padronização dos ambientes de passagem de mensagem através de uma especificação para o desenvolvimento de aplicações paralelas.

Uma falha em uma aplicação paralela pode representar altos custos [53], portanto, a sua verificação e validação é uma atividade fundamental. A atividade de teste é uma técnica de verificação e validação de programas. Sua aplicação visa assegurar um nível de qualidade aceitável ao software em desenvolvimento através da descoberta de erros.

O ideal seria testar o programa para todas as entradas possíveis, porém, isso é inviável. Portanto, o usuário deve concentrar seus testes apenas nos casos com maior probabilidade de revelar erros. Para atingir esse objetivo, o usuário aplica diferentes técnicas e critérios de teste.

As técnicas de teste definem princípios que baseiam os critérios de teste. Critério de teste é um predicado de teste a ser satisfeito para auxiliar a geração de casos de teste [43]. Os critérios baseiam-se nos requisitos funcionais, nos erros mais comuns cometidos pelos programadores ou na estrutura do código. Enquanto os critérios funcionais procuram exercitar todos os requisitos funcionais do software, os critérios estruturais destacam-se pela cobertura do código, pois fornecem medidas que indicam diretamente quanto do software foi testado. O uso de ferramentas que apóiem o processo de teste é imprescindível para uma aplicação efetiva dos critérios de teste [21], visto que essa atividade demanda muito esforço e está propensa a erros se conduzida manualmente.

Os critérios de teste para programas seqüenciais não são adequados para programas paralelos, pois, estes apresentam características não encontradas em programas seqüencias, tais como, a comunicação e sincronização entre tarefas e o não-determinismo. Por isso, os principais critérios de teste utilizados em programas seqüenciais tiveram que ser adaptados para programas paralelos.

A pesquisa de técnicas e critérios de teste para programas paralelos está concentrada no paradigma concorrente de memória compartilhada, sendo que existem ferramentas de teste para esse paradigma. São poucos, porém, os trabalhos relacionados com testes para o paradigma de programação paralela por passagem de mensagem. Dentre esses trabalhos, destaca-se o de Vergilio et al. [60], do grupo de pesquisa do projeto ValiPVM, que propõe critérios estruturais de teste para paradigma de passagem de mensagem.

O grupo de pesquisa do projeto ValiPVM, apoiado pelo CNPq e formado por pesquisadores da Universidade Federal do Paraná e da Universidade Estadual de Ponta Grossa, tem por objetivo estudar e desenvolver novas técnicas e critérios na área de teste de aplicações paralelas em ambientes de passagem de mensagem e a especificar uma ferramenta [47] que automatize a aplicação dos critérios de teste definidos e que possa ser aplicada em problemas reais.

Existem ferramentas para o auxiliar o teste de programas paralelos, porém, a maioria dessas ferramentas possibilita apenas a visualização, depuração ou controle dos programas paralelos sem aplicar critérios de teste. Alguns exemplos destas ferrametas são: TDC Ada [50], ConAn (*Concurrency Analyser*) [32], Xab [3], MDB [12], Paragraph [24],

Della Pasta (*DELaware PArallel Software Testing Aid*) [65], STEPS (*Structural TESting of Parallel Software*) [33, 40], Astral (Ambiente de Simulação e Teste de pRogramas parALelos) [38], Umpire [61], Visit (*Visualize it!*) [27], XPVM [28] e XMPI [56]. Não existe conhecimento sobre ferramentas que apliquem critérios estruturais de teste para o ambientes MPI.

1.2 Motivação

Dado o contexto acima, os principais pontos que motivam este trabalho são os seguintes:

1. A importância do paralelismo e do padrão MPI.
2. A necessidade de testar programas paralelos é evidente para sua verificação e validação.
3. A aplicação dos critérios estruturais de teste, especialmente desenvolvidos para programas paralelos que utilizam o paradigma de passagem de mensagem, é um auxílio importante à atividade de teste, pois permite a quantificação dessa atividade oferecendo medidas de cobertura.
4. A necessidade de ferramentas que automatizem a atividade de teste e permitam a aplicação prática de critérios de teste no ambiente de passagem de mensagem.
5. A existência de ferramentas de teste para programas paralelos. Porém, nenhuma delas apóia o teste de programas paralelos para o ambiente MPI.

1.3 Objetivo

Este trabalho tem por objetivo a implementação de uma ferramenta para teste de programas paralelos desenvolvidos na linguagem C, que utilizam o padrão de passagem de mensagem MPI [36, 37], denominada ValiMPI. Essa ferramenta implementa os critérios estruturais, baseados em fluxo de controle e fluxo de dados, para programas paralelos em ambiente de passagem de mensagem propostos por Vergilio et al. [60], no contexto do projeto ValiPVM. Essa ferramenta permite a execução do programa paralelo, de maneira determinística ou não-determinística, fornece a lista dos elementos requeridos pelos critérios de teste e realiza a análise de adequação do conjunto de testes fornecendo a medida de cobertura.

1.4 Organização do Trabalho

O restante desta dissertação está organizada da seguinte forma: nos Capítulos 2 e 3 são introduzidos, respectivamente, conceitos básicos de teste de software e paralelismo. No Capítulo 4 revisa-se a bibliografia, relacionando esta dissertação com outros trabalhos e introduzem-se os critérios de teste para o paradigma de passagem de mensagem. No Capítulo 5 descrevem-se a arquitetura e aspectos de implementação da ferramenta ValiMPI. No Capítulo 6 apresenta-se o resultado de experimentos realizados com a ferramenta ValiMPI. No Capítulo 7 estão as conclusões, as contribuições e possíveis melhorias às pesquisas realizadas. Esta dissertação possui dois apêndices: no Apêndice A são ilustrados os principais algoritmos da biblioteca ValiMPI e no Apêndice B a sintaxe de linha de comando da ferramenta é explicada.

2 Teste de Software

Uma das metas da Engenharia de Software é a produção de software de qualidade, pois os custos associados aos defeitos de software [53] justificam, na maioria dos casos, uma atividade de teste cuidadosa e bem planejada. A correção de defeitos em software no ambiente de produção é reconhecidamente mais custosa do que nas fases de desenvolvimento [21, 31], já que pode envolver prejuízos de produção, financeiros e humanos.

Para que o software atinja um grau de qualidade aceitável são necessárias atividades de garantia de qualidade, tais como verificação e validação. A atividade de teste de software é uma técnica de verificação e validação e representa a última revisão de especificação, projeto e codificação [41]. Tal atividade é constituída pelo projeto de casos de teste, sua execução e comparação dos resultados obtidos com os resultados esperados, visando fornecer evidências da confiabilidade do software.

2.1 Objetivo

O objetivo da atividade de teste pode ser descrito da seguinte forma [39]: a atividade de teste é o processo de execução de um programa com a intenção de descobrir erros. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar algum erro ainda não descoberto. E um teste bem-sucedido é aquele que revela algum erro ainda não descoberto.

Num ambiente ideal, dado um conjunto de casos de teste, todos os defeitos de um programa seriam revelados. Entretanto, não existe um algoritmo capaz de gerar o conjunto de teste ideal para um determinado programa, pois isto implicaria na geração de um conjunto igual ao de todas as entradas possíveis para o programa.

Testar um programa com todos os valores possíveis do domínio de entrada é impraticável devido às restrições de custo e prazo. É necessário, portanto, determinar quais casos de teste usar, maximizando a quantidade de erros encontrados, de forma que o

número de casos de teste não seja tão grande ao ponto de ser impraticável. Sendo assim, a atividade de teste não pode mostrar a ausência de erros, ela só pode mostrar se defeitos de software estão presentes [41].

2.2 Terminologia

Alguns termos empregados no contexto deste trabalho são usados e encontrados com diversas definições na literatura. Outros ainda, são de certa dificuldade de distinção entre eles. Para que este trabalho possa ser entendido corretamente sem que restem dúvidas sobre esses conceitos, é importante que eles sejam esclarecidos e definidos *a priori*.

Falha (*Failure*): produção de uma saída incorreta com relação à especificação [26].

Defeito (*Fault*): um problema no programa ou na especificação [26], comumente chamado de *bug*.

Erro (*Error*): um resultado incorreto é obtido [26].

Engano (*Mistake*): ação humana que produz um resultado incorreto.

Verificação: conjunto de atividades que garante que o software implementa corretamente uma função específica [41, p. 836].

Validação: conjunto de atividades que garante que o software construído segue as exigências do cliente [41, p. 836].

2.3 Fases

Sob o ponto de vista do custo de desenvolvimento de um sistema, a atividade de teste é a mais dispendiosa, tanto em recursos quanto em prazo [39]. Para cada fase de desenvolvimento do sistema há uma fase de teste correspondente. As fases de teste são: unidade, integração e sistema [41].

O teste de unidade concentra-se na verificação da menor unidade de projeto de software: o módulo, e procura identificar erros de lógica e implementação. O teste de integração verifica a interação entre os módulos. O teste de sistema procura assegurar que o software e os demais elementos do sistema funcionem adequadamente quando combinados.

2.4 Técnicas

Uma técnica de teste define os princípios que baseiam os critérios de testes. Na tentativa de reduzir os custos associados à atividade de teste, foram propostos critérios para auxiliar o usuário[1] na seleção, avaliação e aplicação de casos de teste em um programa, uma vez que executar o programa para todas as entradas do seu domínio raramente é possível.

Critério de teste é um predicado a ser satisfeito para auxiliar a geração de casos de teste e encerrar a atividade de teste, ou seja, dizer se o programa foi testado o suficiente [18]. A utilização de critérios é bastante importante, pois além de guiar o usuário na etapa de seleção de casos de teste, oferece medidas que quantificam a atividade de teste.

A atividade de teste é considerada encerrada quando todos os elementos requeridos por um dado critério de teste tiverem sido cobertos. Os critérios que orientam a escolha dos casos de teste variam de acordo com a técnica utilizada.

As técnicas de testes são as seguintes:

Funcional (ou caixa preta): estabelece casos de teste a partir dos requisitos funcionais de software. Possibilita que o usuário derive dados de teste que exercitem completamente todos os requisitos funcionais do programa. São exemplos de critérios para a técnica funcional [41]: análise do valor limite, particionamento em classes de equivalência e grafo de causa e efeito.

Estrutural (ou caixa branca): utiliza o código fonte do software para estabelecer os casos de teste. Um grafo de fluxo é associado ao programa e alguns caminhos devem ser executados para que o critério estabelecido seja satisfeito. São exemplos de critérios para a técnica estrutural: baseado na complexidade [35], baseado em fluxo de controle [20] e baseado em fluxo de dados [43, 18, 34].

Baseada em erros: utiliza informações sobre o processo de desenvolvimento de software e considera os defeitos mais comuns cometidos pelos programadores para estabelecer os requisitos de teste. Semeadura de erros e análise de mutantes [13] são exemplos de critérios de teste baseados em defeitos.

As técnicas são consideradas complementares [34], porque podem revelar diferentes tipos de defeitos, e devem ser utilizadas em conjunto a fim de realizar uma atividade de teste

[1] "Usuário" no contexto desta dissertação refere-se à pessoa que realiza os testes.

de boa qualidade.

2.5 Critérios Estruturais

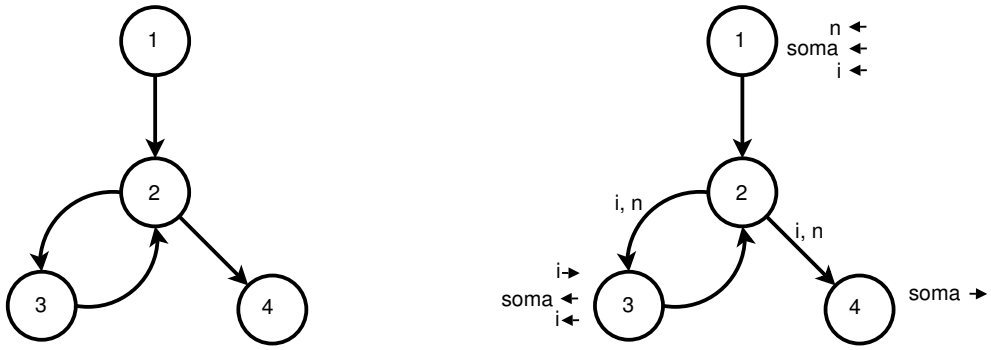
Critérios estruturais de teste são baseados na estrutura interna do programa, ou seja, seu código fonte. É definido um conjunto de elementos de software que deve ser executado para que se atinja uma cobertura mínima do critério. Estes elementos são os componentes do programa requeridos pelo critério e que devem ser testados. A cobertura é a medida que avalia quantos elementos foram testados [43, 34].

A estrutura interna do programa é representada por um grafo de fluxo de controle (GFC). Esse grafo é dirigido e possui um único nó de entrada e um único nó de saída. Cada nó representa uma seqüência de comandos que são sempre executados como um bloco de comandos e cada aresta representa uma transferência de controle entre esses blocos.

São dois os principais tipos de critérios estruturais: os baseados em fluxo de controle e os baseados em fluxo de dados. O grafo de fluxo de dados, também chamado de grafo definição-uso, diferencia-se do grafo de fluxo de controle por possuir informações sobre nós com definições e conseqüentes usos das variáveis do programa. Os grafos das Figuras 2.1(a) e 2.1(b) ilustram, respectivamente, um grafo de fluxo de controle e um grafo de fluxo de dados para o programa da Listagem 2.1.

Listagem 2.1: Programa exemplo: $\sum_{i=1}^{n} i^2$

```
void main()
{
    float soma;
    int i, n;
5   scanf("%d", &n);                /* nó  1 */
    soma = 1;                       /* nó  1 */
    for (i = 0; i < n; i++)         /* nós 1, 2 e 3 */
        soma = soma + pow(i, 2);    /* nó  3 */
    printf("%f", soma);             /* nó  4 */
10 }
```



(a) Grafo de fluxo de controle

(b) Grafo de fluxo de dados

Figura 2.1: Grafos de fluxo para a Listagem 2.1

Os critérios baseados em fluxo de controle são os seguintes:

Todos-nós: requer que todos os nós sejam executados pelo menos uma vez.

Todas-arestas: requer que todas as arestas (transferência de controle) sejam executadas pelo menos uma vez.

Todos-caminhos: requer que todos os caminhos possíveis do programa sejam executados.

Dentre os critérios de fluxo de controle, o mais utilizado é o todas-arestas, entretanto, mesmo para programas pequenos, esse critério não revela a presença de erros simples e triviais [43, 34], fato que motivou a introdução dos critérios baseados em fluxo de dados. O critério todos-caminhos é, em geral, impraticável, pois corresponde ao teste exaustivo.

Os critérios baseados em fluxo de dados requerem que sejam testadas interações que envolvam definições de variáveis e usos subseqüentes destas definições. Sendo assim, os casos de teste devem exercitar usos ou potenciais usos dessas variáveis.

Os seguintes conceitos relacionados com variáveis e caminhos são necessários para o entendimento dos critérios baseados em fluxo de dados:

Definição de variável: ocorre ao se armazenar um valor em uma posição de memória, e é representado no grafo de fluxo de dados pelo nome da variável sucedido pelo símbolo $\leftarrow$.

Uso computacional (ou c-uso): é a referência ao valor de uma variável que afeta diretamente o resultado de uma computação, e é representado no grafo de fluxo de

dados pelo nome da variável sucedido pelo símbolo $\rightarrow$.

Uso predicativo (ou p-uso): é a referência ao valor de uma variável que afeta diretamente o fluxo de controle, e é representado no grafo de fluxo de dados pelo nome da variável adjacente à uma aresta.

Um caminho da forma $(i, n_1, \ldots, n_m, j)$ é dito livre de definição se há definição da variável x em i e não há definição de x nos nós $n_1 \ldots n_m$. Um caminho é dito simples se todos os nós, exceto possivelmente o primeiro e o último, são distintos. Um caminho é dito livre de laços se todos os seus nós são distintos.

Alguns critérios baseados em fluxo de dados são descritos abaixo:

Todos-c-usos: requer que todas as associações c-uso sejam exercitadas. Uma associação c-uso é uma tripla $\langle i, j, x \rangle$ onde o nó $x \in def(i)$ e $j \in dcu(x, i)$. O $def(i)$ é o conjunto de variáveis que são definidas no nó i, e o $dcu(x, i)$ é o conjunto de nós j, tal que, $x \in def(i)$, x tem um uso computacional em j e não há redefinição de x de i para j.

Todos-p-usos: requer que todas as associações p-uso sejam exercitadas. Uma associação p-uso é uma tripla $\langle i, (j, k), x \rangle$ onde o nó i contém uma definição de x e $(j, k) \in dpu(x, i)$. O $dpu(x, i)$ é o conjunto de arestas (j, k), tal que, $x \in def(i)$, x tem um uso predicativo em (j, k) e não há redefinição de x de i para (j, k).

Todos-usos: requer que todas as associações c-uso e todas as associações p-usos sejam exercitadas.

Todos-du-caminhos: requer que todos os du-caminhos[2] de i para j c.r.a x (com relação a x) para cada $j \in dpu(x, i)$ e todos os du-caminhos de i para (j, k) c.r.a x para cada $(j, k) \in dpu(x, i)$ sejam exercitados. Um caminho $(n_1, \ldots, n_j, n_k)$ é um du-caminho c.r.a x se x é definida em n_1 e n_k tem um c-uso de x e o caminho $(n_1, \ldots, n_j, n_k)$ é simples e livre de definição c.r.a x, ou (n_j, n_k) tem um p-uso de x e o caminho $(n_1, \ldots, n_j)$ é livre de definição c.r.a x e livre de laços.

Todos-potenciais-usos: requer que todas as associações potenciais c-uso e todas as associações potenciais p-uso sejam exercitadas. As associações potenciais c-uso e potenciais p-uso diferem das associações c-uso e p-uso por não requererem o uso explícito da variável x no nó j ou no aresta (j, k).

[2] O mesmo que caminhos definição-uso.

A Tabela 2.1 apresenta os elementos requeridos, para o exemplo da Figura 2.1, em relação aos critérios estruturais.

Tabela 2.1: Elementos requeridos para o exemplo da Figura 2.1

Critério	**Elementos requeridos**
todos-nós	1, 2, 3, 4
todas-arestas	(1,2) (2,3) (3,2) (2,4)
todos-c-usos	$\langle 1, 4, \{soma\}\rangle\langle 1, 3, \{soma, i\}\rangle$ $\langle 3, 3, \{soma, i\}\rangle\langle 3, 4, \{soma\}\rangle$
todos-p-usos	$\langle 1, (2, 3), \{n, i\}\rangle\langle 1, (2, 4), \{n, i\}\rangle$ $\langle 3, (2, 3), \{i\}\rangle\langle 3, (2, 4), \{i\}\rangle$
todos-usos	$\langle 1, 4, \{soma\}\rangle\langle 1, 3, \{soma, i\}\rangle$ $\langle 3, 3, \{soma, i\}\rangle\langle 3, 4, \{soma\}\rangle$ $\langle 1, (2, 3), \{n, i\}\rangle\langle 1, (2, 4), \{n, i\}\rangle$ $\langle 3, (2, 3), \{i\}\rangle\langle 3, (2, 4), \{i\}\rangle$
todos-du-caminhos	1 2 4, 1 2 3, 3 2 4, 3 2 3
todos-potenciais-usos	$\langle 1, 2, \{soma, i, n\}\rangle\langle 1, 3, \{soma, i, n\}\rangle$ $\langle 1, 4, \{soma, i, n\}\rangle\langle 3, 2, \{soma, i\}\rangle$ $\langle 3, 3, \{soma, i\}\rangle\langle 3, 4, \{soma, i\}\rangle$ $\langle 1, (1, 2), \{soma, i, n\}\rangle\langle 1, (2, 3), \{soma, i, n\}\rangle$ $\langle 1, (2, 4), \{soma, i, n\}\rangle\langle 1, (3, 2), \{n\}\rangle$ $\langle 3, (2, 3), \{soma, i\}\rangle\langle 3, (2, 4), \{soma, i\}\rangle$ $\langle 3, (3, 2), \{soma, i\}\rangle$

A técnica estrutural de teste apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes às atividades de teste enquanto estratégia de validação [34].

Uma dessas limitações é a correção coincidente. O programa, mesmo que incorreto, pode apresentar, coincidentemente, um resultado correto para um item particular de dado. Essa limitação deve ser considerada como fundamental para qualquer estratégia de teste.

Outra importante limitação é a dos caminhos não executáveis. Um caminho é dito não executável se não existir um conjunto de dados de teste que causem a sua execução [18]. Se todos os caminhos que cobrem um elemento requerido por um critério estrutural forem não executáveis, o elemento é dito não executável, sendo impossível satisfazer o critério, ou seja, obter uma cobertura de 100% dos elementos requeridos. Nesse caso, a partir de um critério estrutural C, deriva-se um critério C*, chamado executável, que requer somente elementos executáveis e pode ser sempre satisfeito [18].

Não existe algoritmo de propósito geral que determine automaticamente se um caminho é não executável [18]. Essa é uma questão indecidível.

2.5.1 Ferramentas

A aplicação de critérios estruturais de teste sem apoio de uma ferramenta automatizada é limitada a programas muito simples [21]. De acordo com a fase de teste, a ferramenta gera os elementos requeridos, os quais o usuário tentará cobrir com a entrada de dados de teste. O usuário entra dados de teste enquanto for possível aumentar a cobertura dos elementos requeridos para os critérios desejados.

2.5.1.1 A Ferramenta de Teste *PokeTool*

A ferramenta *PokeTool* [9] apóia a aplicação dos critérios estruturais, baseados em fluxo de controle e baseados em fluxo de dados, para funções escritas na linguagem C. O seu uso é bastante simples [6] e o fluxo de informações da ferramenta é ilustrado pela Figura 2.2.

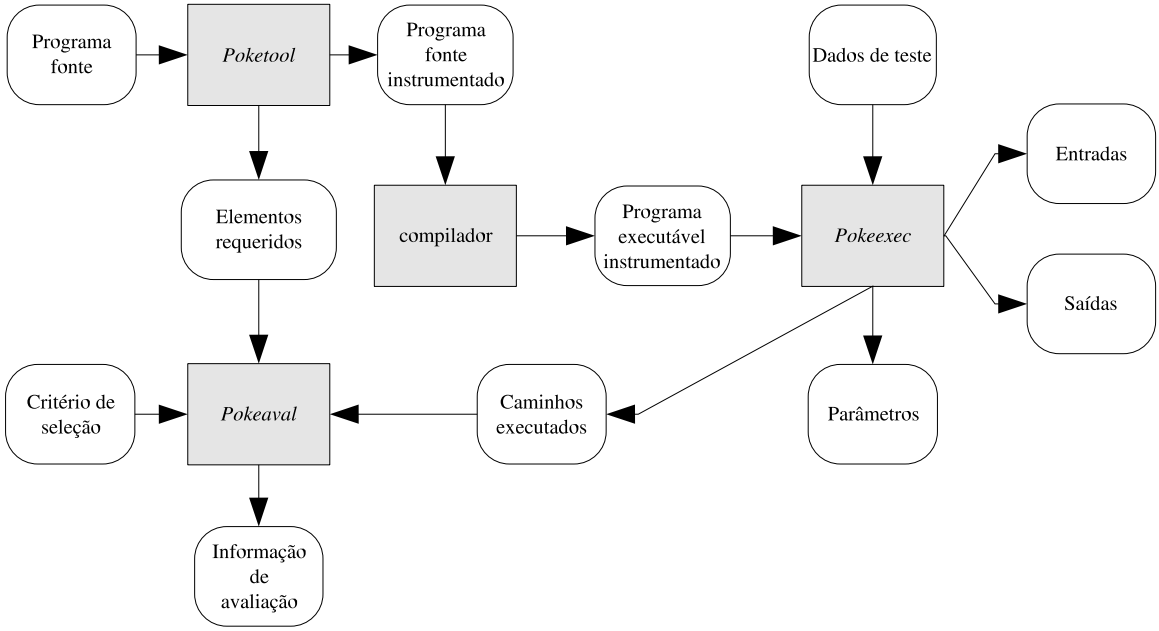


Figura 2.2: Fluxo de informações da *PokeTool*

A *PokeTool* é compreendida por três programas: poketool, pokeexec e pokeaval. O programa poketool gera a lista de elementos requeridos e uma versão modificada do programa fonte original, chamada de programa instrumentado. O programa instrumentado, depois de compilado, é executado pelo programa pokeexec, que armazena os dados de teste e a informação relativa aos elementos cobertos pelos mesmos. O programa pokeaval, confronta os elementos requeridos por um critério de seleção escolhido pelo usuário com os elementos cobertos pelos casos de teste, informando ao usuário quais elementos ainda

não foram cobertos. Isto possibilita ao usuário realizar novos testes a fim de maximizar a cobertura para o critério escolhido.

2.5.1.2 IDeL

A técnica de instrumentação é freqüentemente usada na engenharia de software para diversos propósitos, como por exemplo: traço de execução de programas ou especificação, engenharia reversa e análise de cobertura de critérios de teste. A instrumentação consiste na geração de um grafo que representa o programa e na incorporação de comandos de saída ao código fonte, que não alteram a semântica do programa e que indicam em qual ponto do grafo o programa se encontra em tempo de execução. Al-ladam adverte que a instrumentação poderia alterar o comportamento de aplicações de tempo real [1].

Por ser um processo lento e muito sujeito a erros quando feito manualmente, a instrumentação é automatizada por um programa instrumentador. A maioria dos instrumentadores é específica para uma linguagem ou domínio, o que torna difícil o reuso e a evolução dos softwares instrumentados.

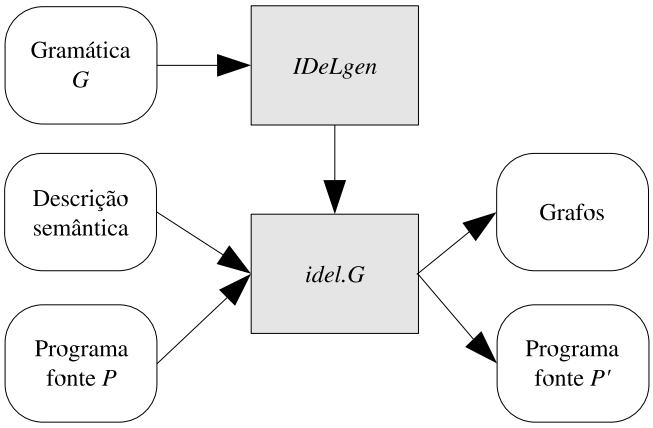


Figura 2.3: Diagrama de execução da IDeLgen e do instrumentador

A IDeL (*Instrumentation Description Language*) [45, 44] é uma meta-linguagem orientada à instrumentação, que suporta a descrição de ambas tarefas do processo de instrumentação: a derivação do grafo da estrutura e a inclusão dos comandos. A meta-linguagem é instanciada por uma gramática livre de contexto para uma linguagem específica (G), da qual se gera o programa instrumentador ($idel.G$) através do "compilador" IDeLgen (*IDeL Generator*). O programa instrumentador, por sua vez, recebe um programa fonte (P) e a descrição semântica da instrumentação, gerando o programa fonte instrumentado

(*P'*) e a representação de cada unidade de teste em forma de grafo. A Figura 2.3 ilustra a execução da IDeLgen e do instrumentador *idel.G*.

2.6 Considerações Finais

Este capítulo abordou conceitos básicos sobre teste de software para programas seqüenciais que serão usados ao longo desta dissertação.

O próximo capítulo apresenta conceitos básicos sobre paralelismo, ao passo que o Capítulo 4 apresentará teste de software paralelo.

A ValiMPI, detalhada no Capítulo 5, foi inspirada na *Poketool*. A IDeL, usando um subconjunto da gramática da linguagem C, é utilizada pela ValiMPI na fase de instrumentação.

3 Paralelismo

Com o aumento do poder computacional é tentador se imaginar que um dia os computadores serão rápidos o suficiente e a demanda por maior poder computacional será satisfeita. Entretanto, historicamente quando uma tecnologia satisfaz as aplicações conhecidas, novas necessidades surgem demandando o desenvolvimento de novas tecnologias [16]. A programação paralela tem por objetivo resolver problemas complexos rapidamente, decompondo o problema entre os processadores disponíveis [2, p. 5].

Tradicionalmente, o desenvolvimento de aplicações paralelas é motivado pela simulação numérica de sistemas complexos como: tempo, clima, mecânica dos fluídos, circuitos eletrônicos, processos de manufatura e reações químicas. Atualmente, a demanda por sistemas computacionais paralelos vem também de aplicações para a indústria de entretenimento, inteligência artificial, processamento de imagens, bioinformática, bases de dados paralelas, simulação do mercado financeiro, entre outros [51, 60].

A demanda computacional, aliada à popularização dos *clusters* de computadores e a padronização de bibliotecas para programação paralela multiplataforma, favorecem o investimento no desenvolvimento de aplicações paralelas.

3.1 O Modelo Seqüencial

Tradicionalmente os programas de computador são escritos visando o modelo de máquina de von Neumann, ou seja, uma unidade de processamento (CPU) que executa seqüências de leitura e escrita em uma unidade de armazenamento (memória).

Este modelo permitiu que o estudo da arquitetura de computadores procedesse de maneira independente do estudo de algoritmos e linguagens de programação. Desta forma, o programador pode desenvolver seus algoritmos numa linguagem que abstraia a máquina de von Neumann, ao invés de fazê-lo em outra dependente do *hardware* utilizado.

3.2 O Modelo Paralelo

Um computador paralelo é um conjunto de processadores aptos a trabalharem cooperativamente para resolver um problema computacional [16]. Esta definição é suficientemente abrangente para incluir supercomputadores paralelos com centenas ou milhares de processadores, redes de estações de trabalho, estações multiprocessadas e sistemas embarcados.

O modelo conceitual de máquina paralela [25, p. 18] é uma abstração de um computador paralelo do ponto de vista do programador. Tal modelo caracteriza as capacidades de um computador paralelo que são fundamentais à computação paralela. Essa abstração não implica em informações estruturais, tais como, o número de processadores e a estrutura de comunicação interprocessador, mas captura os custos relativos da computação paralela.

3.2.1 Taxonomia

Apesar de todos sistemas paralelos serem constituídos por vários elementos de processamento (*processing elements* ou PEs), há várias maneiras de organizar o hardware, especialmente em termos da interconexão entre processadores e sua comunicação.

Alguns esquemas de classificação foram propostos para sistemas com vários PEs, dos quais o mais freqüentemente citado é, provavelmente, a taxonomia de Flynn [15]. Flynn usa o conceito de fluxo para descrever a estrutura de uma máquina. Um fluxo é uma seqüência de dados ou instruções.

Segundo esta classificação, os computadores dividem-se em quatro categorias:

SISD (*Single Instruction stream, Single Data stream*) corresponde ao mono-processador tradicional (computador de von Neumann).

SIMD (*Single Instruction stream, Multiple Data streams*) vários PEs, cada um com sua própria memória, executam a mesma instrução. Esse modelo é adequado apenas para algumas tarefas altamente especializadas, caracterizadas pelo alto grau de regularidade.

MISD (*Multiple Instruction streams, Single Data stream*) nenhum computador existente encaixa-se nesta categoria.

MIMD (*Multiple Instruction streams, Multiple Data streams*) vários PEs, executam diferentes conjuntos de instruções em diferentes conjuntos de dados.

A maioria dos computadores paralelos existentes são classificados como MIMD. Podemos dividir os computadores MIMD em dois grupos: aqueles com memória distribuída, chamados de *multicomputadores* e aqueles com memória compartilhada, chamados *multiprocessadores*.

3.2.1.1 Multicomputador

Um multicomputador [16], Figura 3.1, é formado por nós interconectados, cada qual com pelo menos um processador e memória própria. Toda comunicação e sincronização é feita através de mensagens, que são usadas para a comunicação entre os nós, lendo e escrevendo em memória remota. Em uma interconexão ideal, o custo de uma mensagem entre dois nós independe da sua localização e do tráfego, mas não do tamanho da mensagem. Na prática o custo de acesso da memória local é menor e portanto é desejável que um programa acesse mais freqüentemente a memória local do que a remota, uma propriedade chamada *localidade*.

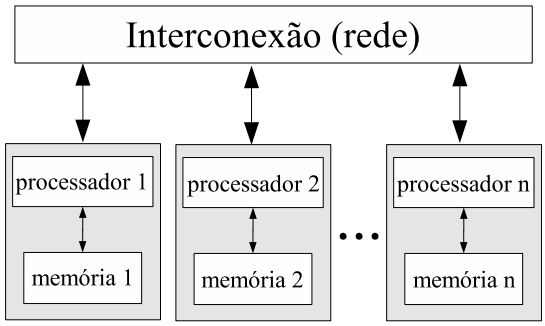


Figura 3.1: Arquitetura do multicomputador

Um *cluster* de computadores [2, p. 475] é um caso particular de multicomputador. Formado por uma coleção de estações de trabalho ou computadores pessoais interconectados, tipicamente, por uma rede local, cada nó do *cluster* pode ter um ou mais processadores, todavia todos os nós trabalham em conjunto como um único recurso. Devido à boa relação custo/desempenho os *clusters* são bastante populares.

3.2.1.2 Multiprocessador

Em um multiprocessador [16], Figura 3.2, todos os processadores compartilham o acesso a uma mesma memória através de um único espaço de endereçamento. O acesso à

memória *cache* associada a cada processador é muito mais rápido que o acesso à memória comum, o que torna importante a localidade.

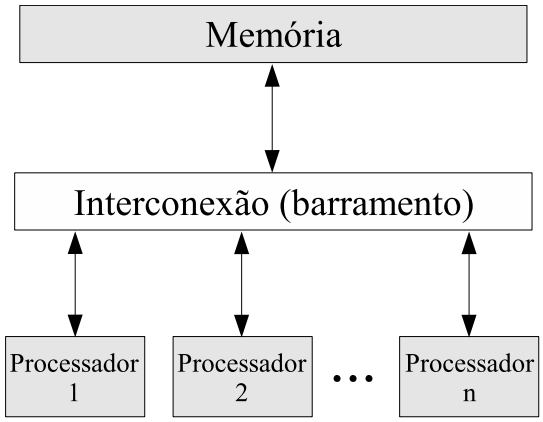


Figura 3.2: Arquitetura do multiprocessador

3.2.2 Programação Paralela

A programação seqüencial é formada por instruções executadas seqüencialmente, desvios condicionais e incondicionais, estruturas de repetição e subrotinas. Para determinadas aplicações esses recursos de programação são insuficientes para a computação de resultados em um intervalo de tempo aceitável.

A programação paralela tem por objetivo otimizar o desempenho das aplicações. Esse objetivo é alcançado através da divisão da computação em subtarefas menores, nas quais cada uma resolva uma porção do problema. Para efetuar essa divisão necessita-se de recursos adicionais que são providos por uma linguagem de programação paralela ou uma biblioteca de funções para programa paralela escrita em uma linguagem para programação seqüencial. Uma linguagem de programação ou biblioteca de funções para programação paralela deve privilegiar a concorrência, localidade, escalabilidade, portabilidade, modularidade, desempenho, além de possibilitar a implementação do maior número possível de algoritmos paralelos. Estes conceitos são explicados a seguir:

Concorrência é a capacidade de executar processos[1] simultaneamente, ou com aparência de simultaneidade (pseudo-paralelismo) [16].

Paralelismo ocorre quando há mais de um processo executando no mesmo intervalo de tempo em sistemas com mais de um processador [16].

[1]Entende-se por processo um programa em execução, que consiste de um programa executável, seus dados, contador de instruções, registradores e todas as informações necessárias para sua execução [52].

Localidade indica que o custo de uma mensagem entre dois nós depende da sua localização e do tráfego [16]. Acessar a memória local é geralmente mais rápido do que a remota.

Escalabilidade é a capacidade de um sistema aumentar o seu desempenho quando há um aumento nos seus recursos (tipicamente hardware) [25, p. 6].

Portabilidade é a capacidade de levar (compilar e executar) um programa de uma plataforma à outra, o que preserva o investimento feito na implementação [41].

Modularidade é a capacidade de dividir o problema em problemas menores e autocontidos [41].

Desempenho em paralelismo corresponde ao tempo levado para executar um programa, considerando-se o tempo gasto na computação, comunicação e gerência dos processos paralelos [25].

Desse modo, programas concorrentes são tipicamente formados por vários processos que interagem entre si para gerar um resultado. Segundo Almasi e Gottlieb [2], três fatores são fundamentais para a execução paralela:

- Definir um conjunto de subtarefas para serem executadas em paralelo.
- Capacidade de iniciar e finalizar a execução das subtarefas.
- Capacidade de coordenar e especificar a interação entre as subtarefas.

3.2.3 Paradigmas de Programação Paralela

Como a arquitetura do computador paralelo pode variar consideravelmente, diferentes técnicas de programação devem ser aplicadas a diferentes computadores. Multicomputadores são freqüentemente associados ao paradigma de passagem de mensagem, enquanto os multiprocessadores estão mais próximos do paradigma de memória compartilhada. Entretanto, programas desenvolvidos para multicomputadores também podem ser executados eficientemente em multiprocessadores, pois a memória compartilhada permite a implementação eficiente do paradigma de passagem de mensagem [16].

Os paradigmas de programação paralela mais usados são: tarefas e canais, paralelismo de dados, memória compartilhada e passagem de mensagem. Esses paradigmas são apresentados a seguir.

3.2.3.1 Tarefas e Canais

O paradigma de tarefas e canais [16] considera a computação como um conjunto de tarefas concorrentes que se comunicam através de canais. Uma tarefa encapsula um programa e uma memória local e define um conjunto de portas que funcionam como interface com o meio. Um canal é uma fila de mensagens na qual o remetente posta suas mensagens e da qual o destinatário pode retirar mensagens, esperando bloqueado se não há mensagens disponíveis.

3.2.3.2 Paralelismo de Dados

O paralelismo de dados explora a concorrência que deriva da aplicação da mesma operação em múltiplos elementos de uma estrutura de dados. Um programa neste paradigma é formado por uma seqüência deste tipo de operações. Um algoritmo paralelo para um programa em paralelismo de dados é obtido pela aplicação de técnicas de decomposição do domínio nas estruturas de dados operadas. As operações são então particionadas, geralmente o processador que "detém" um valor é responsável por atualizá-lo.

Compiladores para programas de paralelismo de dados geralmente requerem do programador informações sobre como os dados estão distribuídos entre os processadores. O compilador então, gera automaticamente o código de manipulação e comunicação entre tarefas.

C*, pC++ e HPF (*High Performance Fortran*) são exemplos de linguagens que seguem este paradigma. Um problema desta abordagem é a eficiência do código compilado, seja pelo custo excessivo da comunicação, seja pela presença de gargalos seqüenciais. Os gargalos seqüenciais ocorrem quando um fragmento de código não incorpora paralelismo o suficiente ou quando há paralelismo, porém este não é detectado pelo compilador. Em ambos os casos o fragmento de código não poderá ser executado em paralelo.

3.2.3.3 Memória Compartilhada

Neste modelo, as tarefas compartilham um espaço de memória comum, no qual elas lêem e escrevem de forma assíncrona. Mecanismos como semáforos, *busy-waiting* ou monitores para estabelecer a ordem de execução dos processos e controlar o acesso às regiões críticas.

Uma vantagem deste modelo, do ponto de vista do programador, é que não é necessário

explicitar a comunicação de dados entre processos. Isto pode simplificar o desenvolvimento do programa. Entretanto, entender e gerenciar a localidade torna-se mais difícil, além de dificultar o desenvolvimento de programas determinísticos [16]. *Threads*, *Shared Memory*, Ada e OpenMP são exemplos deste paradigma.

3.2.3.4 Passagem de Mensagem

Este é, provavelmente, o paradigma de programação paralela mais usado atualmente. Ambientes de passagem de mensagem são, em geral, bibliotecas de comunicação que permitem a criação de programas paralelos, ou seja, o programa é escrito em uma linguagem seqüencial, como C ou Fortran, estendida através de uma biblioteca que inclui funções para troca de mensagens entre as tarefas.

Algumas linguagens de programação apoiam o uso desse paradigma no nível da própria linguagem, com primitivas de *send* e *receive*. Ada, através de *rendezvous*, e linguagens baseadas em CSP são exemplos dessas linguagens.

Programas que se utilizam do paradigma de passagem de mensagem criam múltiplas tarefas, as quais encapsulam dados locais. Cada tarefa é identificada, *e.g.* através de um número, e interage com outras tarefas através de mensagens.

Os ambientes de passagem de mensagem foram desenvolvidos inicialmente sem a preocupação com a portabilidade, cada fabricante desenvolveu seu próprio ambiente. Atualmente, existem vários ambientes de passagem de mensagem independentes de máquina, como por exemplo: PVM, MPI, p4, Express e PARMACS. Dois destes ambientes se destacam, o PVM (*Parallel Virtual Machine*) [19] e MPI (*Message Passing Interface*) [36, 37]. MPI é a especificação de um padrão para passagem de mensagem, as principais diferenças entre MPI e PVM são abordadas em [22]. Por ser o foco do presente trabalho, o MPI será descrito detalhadamente a seguir.

3.3 MPI

O padrão MPI foi definido por um fórum formado por universidades, empresas e institutos de diversos países. Este padrão especifica sintaxe e semântica de uma biblioteca de funções disponíveis ao programador de aplicações paralelas nas linguagens seqüenciais C, Fortran e C++. Porém, a maneira como esta especificação deve ser implementada não está definida.

O MPI 1.1 [36] especifica, para as linguagens C e Fortran, a comunicação ponto-a-ponto, a comunicação coletiva, grupos de processos e contexto de comunicação, topologias de processos e interface para *profiling*. O MPI 2 [37] estende a especificação anterior com novas operações coletivas, comunicação unilateral, I/O, criação e gerência de processos, interfaces externas, além de adicionar suporte à linguagem C++.

No MPI uma computação é formada por um ou mais processos que se comunicam através de chamadas de subrotinas de envio e recebimento de mensagens. No MPI 1.1 um número fixo de processos é criado na inicialização e distribuído entre os processadores. O MPI 2 é mais flexível e permite a criação de processos dinamicamente. A comunicação entre as tarefas acontece de duas maneiras. A primeira, chamada de comunicação ponto-a-ponto, utiliza operações de *send* e *receive*. A segunda, chamada de comunicação coletiva, utiliza operações tais como *broadcast*, *gather*, *barrier*, entre outras.

3.3.1 Exemplo

A Listagem 3.1 exemplifica uma forma de comunicação ponto-a-ponto em um programa MPI. Supondo que sejam criados dois processos instanciando este programa, o ambiente MPI é inicializado pela função MPI_Init. A função MPI_Comm_rank obtém o número do processo que a invocou e o atribui à variável myrank. O processo 0 envia o *string* "Hello!" para o processo 1, que o imprime. Posteriormente o ambiente é finalizado.

Listagem 3.1: Exemplo de passagem de mensagem em MPI

```
#include "mpi.h"
#include "mpi_hello.h"
int main(int argc, char *argv[]) {
   char msg[10];
   int myrank;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   if (myrank == 0) {
      strcpy(msg, "Hello!");
      MPI_Send(msg, strlen(msg) + 1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
   } else {
      MPI_Recv(msg, sizeof(msg), MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
      printf("received: %s\n", msg);
   }
   MPI_Finalize();
   return 0;
}
```

3.3.2 Biblioteca de Funções

A Tabela 3.1 mostra algumas funções da biblioteca especificada pelo MPI na linguagem C. O padrão completo possui centenas de funções.

Tabela 3.1: Algumas funções do MPI

MPI_Init	rotina de inicialização, chamada apenas uma vez, antes de qualquer outra rotina MPI.
MPI_Finalize	rotina de finalização, nenhuma outra rotina MPI pode ser chamada depois desta.
MPI_Comm_size	retorna o número de tarefas envolvidas em um contexto.
MPI_Comm_rank	retorna o número da tarefa que invocou a função.
MPI_Send	envia um *buffer* para uma tarefa especificada.
MPI_Recv	espera o recebimento de um *buffer* de outra tarefa.
MPI_Test	avalia a requisição do envio ou recebimento de uma mensagem.

3.3.3 Modos de Comunicação

MPI prevê quatro modos de envio de mensagens ponto-a-ponto entre as tarefas dependendo das necessidades do programador. São eles:

Standard a implementação MPI decide se a mensagem será "bufferizada". Este modo é dito não-local, pois a operação de envio pode depender da postagem de um *receive* correspondente para completar.

Ready a operação de envio da mensagem só inicia se um *receive* correspondente já tenha sido postado no processo destino, caso contrário um erro é gerado. Este modo possibilita melhor desempenho em alguns sistemas.

Synchronous a operação de envio da mensagem completa quando um *receive* correspondente tiver sido postado. Este modo é dito não-local.

Buffered a operação de envio da mensagem completa independentemente da postagem de um *receive* correspondente. É necessário alocar um espaço suficiente para a mensagem em um *buffer* especial antes do envio. Este modo é dito local.

Além dos modos de comunicação, é possível determinar se as operações de *send* e *receive* serão bloqueantes ou não-bloqueantes, a fim de melhorar o desempenho e evitar

Tabela 3.2: Funções e modos de comunicação

modo	**bloqueante**	**não-bloqueante**
standard send	MPI_Send	MPI_Isend
ready send	MPI_Rsend	MPI_Irsend
synchronous send	MPI_Ssend	MPI_Issend
buffered send	MPI_Bsend	MPI_Ibsend
receive	MPI_Recv	MPI_Irecv

deadlocks [22]. Nas operações bloqueantes, a função chamada não poderá retornar até que os dados da mensagem tenham sido armazenados de maneira que o *buffer* de comunicação possa ser sobrescrito. Enquanto que nas operações não-bloqueantes, uma função pode retornar antes de completar, tornando necessário chamar outra função para verificar se a operação foi completada, *e.g.* MPI_Test ou MPI_Wait. *Sends* bloqueantes podem casar com *receives* não-bloqueantes, e vice-versa. As funções de comunicação coletiva são todas bloqueantes.

A cada *send* é associado um envelope, que especifica o destino da mensagem e contém informações que podem ser usadas pela operação de *receive* para selecionar uma mensagem em particular. O envelope é composto pelo número do processo de origem e do processo destino, *tag* e contexto de comunicação. *Tag* [25, p. 669] é um número inteiro que pode ser usado pelo programador para diferenciar as mensagens. Um contexto de comunicação [2, p. 670] provê um "universo de comunicação" separado, mensagens sempre são recebidas no mesmo contexto as quais foram enviadas e mensagens enviadas em contextos diferentes não interferem umas com as outras.

3.4 Determinismo

Programas seqüenciais possuem um comportamento chamado determinístico, ou seja, toda execução de um programa seqüencial com a mesma entrada produzirá sempre a mesma saída. Isso acontece porque os comandos do programa são executados seqüencialmente e os desvios são selecionados deterministicamente, de acordo com os valores de entrada informados.

A maioria dos modelos de programação paralela não garantem a execução determinística. Um programa paralelo é dito não-determinístico quando, para os mesmos dados de entrada, a ordem de execução das suas instruções diferir em alguma das execuções do programa [64].

O não-determinismo pode levar à computação de um resultado final diferente em alguma das execuções do programa [25, p. 86], o que é, em geral, indesejado e pode indicar a presença de um erro, como no exemplo da Listagem 3.2. A Listagem 3.2 ilustra um caso de não-determinismo em MPI, neste exemplo o processo P^2 espera por dados provenientes de quaisquer fontes, não havendo garantia sobre o conteúdo do vetor data no processo P^2 após a execução dos *receives*.

Listagem 3.2: Exemplo de não-determinismo em MPI

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    int myrank;
    float data[2];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0) {
        data[0] = 20.0;
        MPI_Send(data, 1, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
    }
    else if (myrank == 1) {
        data[0] = 30.0;
        MPI_Send(data, 1, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
    }
    else if (myrank == 2) {
        MPI_Recv(&data[0], 1, MPI_FLOAT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
            &status);
        MPI_Recv(&data[1], 1, MPI_FLOAT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
            &status);
        printf("diferença = %.1f \n", data[0] - data[1]);
    }
    MPI_Finalize();
    return 0;
}
```

As duas possibilidades de sincronização deste exemplo estão ilustradas nas Figuras 3.3 e 3.4. Se P^2 sincronizar primeiro com P^0 e depois com P^1, como mostra a Figura 3.3, o programa terá como saída "diferença = −10.0". No caso de P^2 sincronizar primeiro com P^1 e depois com P^0, como mostra a Figura 3.4, o programa terá como saída "diferença = 10.0".

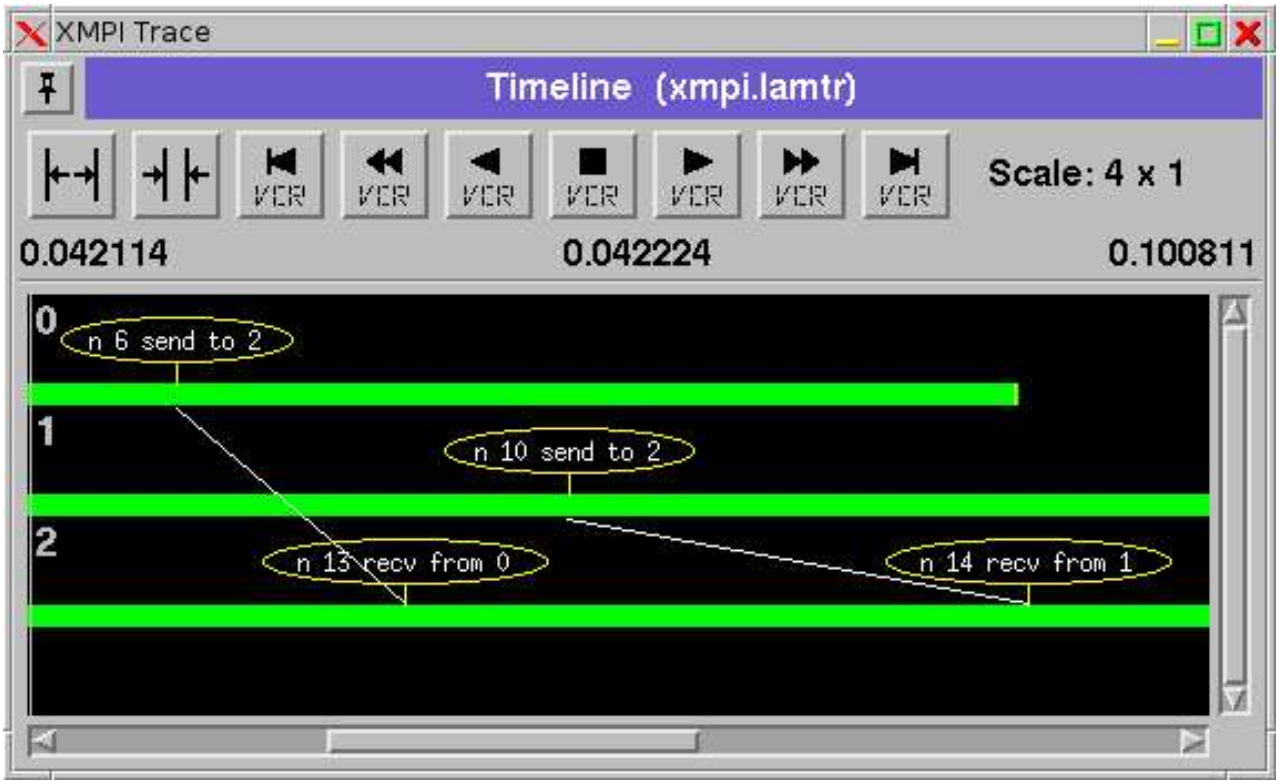


Figura 3.3: P^2 sincroniza com P^0 e P^1 respectivamente

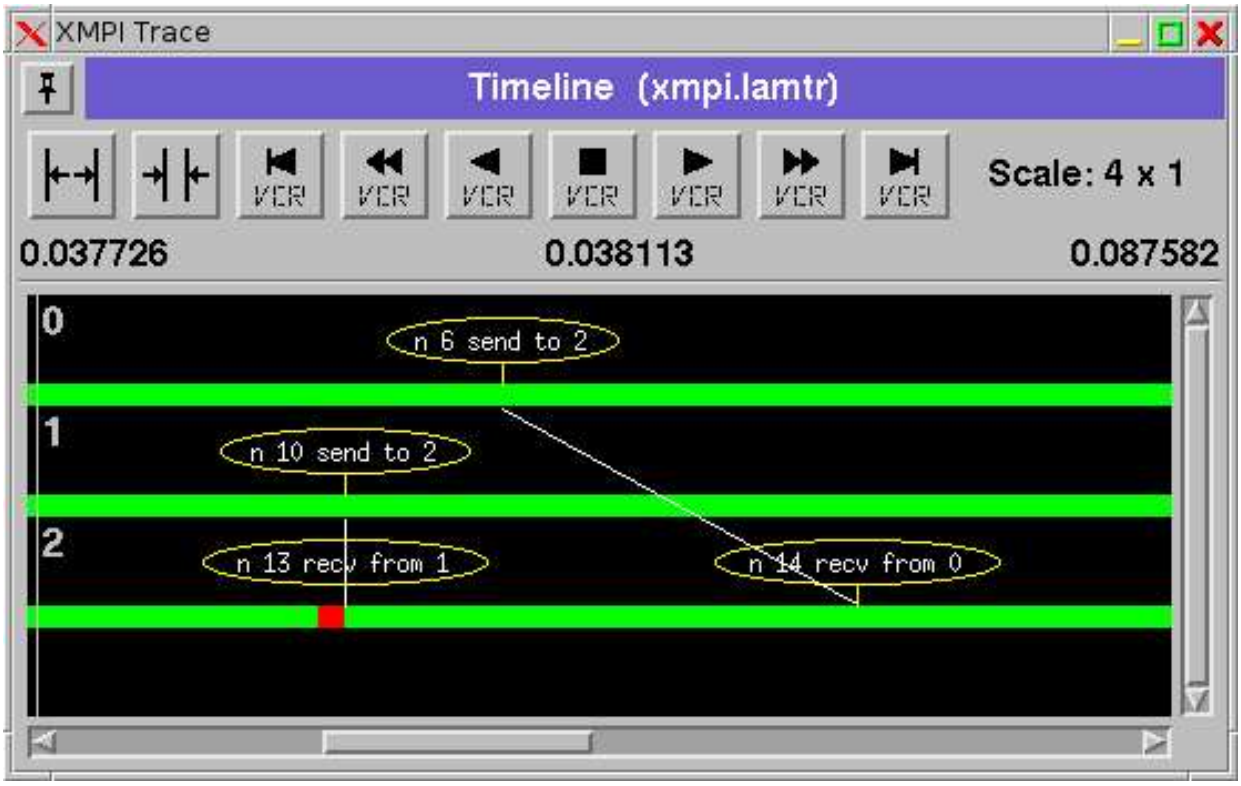


Figura 3.4: P^2 sincroniza com P^1 e P^0 respectivamente

Considera-se o modelo de programação por passagem de mensagem não-determinístico, pois a ordem de chegada das mensagens enviadas por dois processos A e B para um terceiro processo C não está definida. Entretanto, o MPI garante que duas mensagens enviadas de um mesmo processo A, para outro processo B, chegarão na ordem de envio. É responsabilidade do programador assegurar o determinismo da computação quando este é necessário [16].

No modelo de programação tarefa/canal, o determinismo é garantido pela definição de canais separados para diferentes comunicações e pela segurança de que cada canal possui um único escritor e um único leitor. Portanto, o processo C pode distinguir mensagens recebidas do processo A ou B pois elas chegam em canais diferentes. MPI não suporta canais diretamente, mas provê um mecanismo similar. Esse mecanismo é o envelope, que define a origem, o destino, o *tag* e o contexto de comunicação.

A origem, em uma função *receive* especifica que a mensagem será recebida de um processo identificado por um inteiro ou de um processo qualquer, quando usado o valor de origem MPI_ANY_SOURCE. O *tag* fornece um mecanismo adicional para diferenciar mensagens, um *receive* especifica que a mensagem recebida terá um *tag* específico ou um *tag* qualquer, quando o valor deste for MPI_ANY_TAG. A função MPI_Test também pode ser uma fonte de não-determinismo pois, não é possível saber quantas vezes ela deverá ser chamada até que a requisição de recebimento de mensagem que ela testa seja avaliada como recebida.

3.5 Considerações Finais

Este capítulo abordou conceitos básicos sobre paralelismo que serão usados ao longo da dissertação. Os principais paradigmas de programação paralela foram apresentados, com ênfase ao paradigma de passagem de mensagem, um dos mais utilizados atualmente. Uma importante especificação de ambiente de passagem de mensagem é o padrão MPI.

A maioria dos fabricantes de computadores paralelos fornece uma implementação do padrão MPI compatível com seu hardware. Esta dissertação usa a implementação LAM/MPI [7, 48], uma implementação *open source* do MPI 1.1 [36] e de parte do MPI 2 [37], compatível com sistemas operacionais POSIX, desenvolvida inicialmente pelo centro de supercomputação de Ohio. Apesar do uso desta implementação em particular, os resultados obtidos nesta dissertação podem ser aplicados a qualquer implementação do padrão MPI 1.1 ou superior.

Conforme citado no Capítulo 2, a garantia de qualidade de software possui etapas de verificação e validação que incluem o teste de sofware. O próximo capítulo associa o paralelismo ao teste de software, apresentando trabalhos e ferramentas relevantes.

4 Teste de Programas Paralelos

Em geral, o teste de software aplicado a programas paralelos visa identificar erros relacionados à comunicação, paralelismo e sincronização. Entretanto, os erros classificados para programas seqüenciais [4] também devem ser considerados pelos critérios de teste para programas concorrentes.

Como apresentado no Capítulo 2, no contexto da programação seqüencial, foram identificados vários critérios de teste, os quais consideram duas questões importantes para essa atividade: a seleção de casos de teste e a avaliação da adequação dos casos de teste em relação ao programa em teste.

O teste de programas paralelos apresenta dificuldades não encontradas no teste de programas seqüenciais [63, 64]. Apesar dessas dificuldades, o conhecimento adquirido durante a validação de programas seqüenciais foi adaptado ao contexto da programação paralela, dessa forma surgiram trabalhos que estendem as técnicas e os critérios da programação seqüencial [1].

4.1 Dificuldades no Teste de Programas Paralelos

O teste de programas paralelos impõe desafios [63, 64] inexistentes no teste seqüencial. Dentre esses desafios encontram-se dificuldades para:

1. Representar o programa paralelo e capturar as informações pertinentes à atividade de teste.

2. Lidar com o não-determinismo da execução que dificulta a reexecução de um caso de teste.

3. Detectar erros de sincronização, comunicação, fluxo de dados e *deadlock*.

4. Adequar os critérios de teste existentes e algoritmos de geração de dados de teste

utilizados no contexto de programas seqüenciais para o contexto de programas paralelos.

4.2 Tipos de Erros em Programas Paralelos

Dois pontos importantes devem ser considerados para a definição de critérios de teste de programas paralelos: 1) os tipos de defeitos que devem ser evidenciados pelos critérios de teste; e 2) como representar o programa paralelo de modo a obter as informações necessárias para os critérios de teste.

Krawczyk e Wiszniewski [30] apresentam dois tipos de erros relacionados a programas paralelos. Para as definições abaixo, considere $D(p_x)$ como o domínio de entrada de um caminho $p_x(D(p_x))$ formado pelas entradas que exercitam p_x e $C(p_x)$ como a representação da computação de $p_x(C(p_x))$ formado pelas saídas produzidas com a execução de p_x.

1. Erro de observabilidade (*observability error*): ocorre em um programa P se para algum caminho $p_i \in P$ existe outro caminho $p_j \in P$ tal que $D(p_i) = D(p_j)$ e $C(p_i) \neq C(p_j)$. Este erro está relacionado ao ambiente de teste, quando o usuário não pode controlar o comportamento do programa executando em paralelo.

2. Erro de travamento (*locking error*): ocorre em um programa P se para algum caminho $p \in P$, formado pelo conjunto de nós $p = q_0\ q_1 \ldots q_i$ contendo um domínio $D(p_s)$, existe pelo menos um elemento de $D(p_s)$ em que todos os predicados avaliados a partir de q_i são falsos.

4.3 Trabalhos Relacionados

A questão de testes em programas concorrentes foi abordada inicialmente em 1973 por Hansen [23]; porém, até que a programação paralela se tornasse mais popular, poucos trabalhos seguiram essa linha de pesquisa.

Tai [49] generaliza o conceito de Hansen para a reprodução de testes em programas concorrentes; entretanto, não entra em detalhes sobre a sistemática de testes.

Weiss [62] fundamenta a teoria de testes de programas concorrentes. O comportamento da execução de um programa concorrente seria simulado por um conjunto de serializações, que são programas seqüenciais gerados pela junção dos corpos das tarefas.

Todavia, a serialização de tarefas é lenta e o processo de execução longo, especialmente para um grande número de tarefas.

Taylor utiliza o grafo de concorrência [54] obtido a partir da análise estática de programas concorrentes nas linguagens Ada e CSP para representar os possíveis estados de sincronização. Esse grafo sofre da explosão combinatória do número de estados concorrentes, o que limita seu uso a um pequeno número de tarefas. Baseado no grafo de concorrência, Taylor et al. definem critérios de cobertura [55] semelhantes aos baseados em fluxo de dados em programas seqüenciais.

De maneira similar, Chung et al. [10] definem quatro critérios de teste para programas concorrentes para a linguagem Ada, e sua relação de inclusão: 1) *All-Entry-Call*, 2) *All-Possible-Entry-Acceptance*, 3) *All-Entry-Call-Permutation* e 4) *All-Entry-Call-Dependency-Permutation*. Esses critérios focalizam os aspectos de comunicação e sincronização entre as tarefas, expressos por *rendezvous* [2, p. 165].

Yang e Chung [66] propõem: um modelo para representar o comportamento da execução de um programa concorrente, uma estratégia de execução de testes, o processo de teste e uma análise formal para a efetividade da aplicação da análise de caminhos para detectar defeitos em programas concorrentes. O comportamento da execução de um programa é compreendido pela computação (saída produzida e o caminho executado por cada tarefa) e pela comunicação (seqüência de sincronizações e dados trocados em cada sincronização).

A modelagem de um programa concorrente é dada pelos grafos de fluxo de dados (ou *flow graph*) e de sincronização (ou *rendezvous graph*). Cada grafo de fluxo de dados modela o comportamento de execução de uma tarefa e seus possíveis fluxos de execução. O grafo de sincronização modela as possíveis seqüências de sincronização de uma tarefa. Cada execução envolve dois conjuntos de caminhos concorrentes:

- C-PATH, o conjunto de todos os caminhos executados pelas tarefas, no *flow graph*, para uma entrada dada.
- C-ROUTE, o conjunto de seqüências de sincronização, no *rendezvous graph*, possíveis para um C-PATH dado.

Portanto, um caso de teste para o programa P é dado pela 3-tupla (α, σ, δ), que corresponde à computação realizada pela execução de P com a entrada α, atravessando o C-PATH σ e a C-ROUTE δ. O processo de teste examina a corretude de cada C-ROUTE ao longo de todos C-PATHs.

Koppol et al. [29] descrevem um método para selecionar seqüências de testes para programas concorrentes a partir de um LTS (*Labeled Transitions System* ou Sistema de Transições Rotuladas), que é um grafo de estados concorrentes, chamado de grafo de alcançabilidade. O problema da explosão combinatória de estados do grafo é superado através de um grafo reduzido gerado pela análise de alcançabilidade incremental, que consiste em gerar grafos de alcançabilidade sucessivos para subconjuntos de LTSs, reduzindo esse grafo intermediário e usando-o no lugar dos LTSs originais.

Com essa abordagem perdem-se informações necessárias para testes determinísticos. Para não perder estas informações, os autores definem o ALTS (*Annotated Labeled Transitions System* ou Sistema de Transições Rotuladas com Anotações). Um ALTS é um LTS com as anotações necessárias para o teste determinístico. Os autores definem ainda, critérios para seleção de caminhos no ALTS e um algoritmo de redução para o ALTS.

Yang et al. [65] estendem o critério de fluxo de dados todos-du-caminhos para ser usado em programas paralelos. Um Grafo de Fluxo de Programa Paralelo (PPFG — *Parallel Program Flow Graph*) é construído estaticamente a partir do programa, o qual é percorrido para a geração dos du-caminhos. O PPFG corresponde a um grafo de fluxo de dados para cada *thread*, com arestas especiais para indicar a criação e sincronização de *threads*. No PPFG, os vértices são os *statements* do programa e as arestas podem ser de três tipos: 1) arestas *intra-threads* de controle de fluxo, que correspondem às arestas de um grafo de fluxo de controle em um programa seqüencial, 2) arestas de sincronização entre diferentes *threads* e 3) arestas de criação de novos *threads*.

Todos os du-caminhos que possuem definição e uso de variáveis relacionadas ao sincronismo dos *threads* são requisitos de teste a serem executados. Um algoritmo para a geração de todos os du-caminhos e uma ferramenta são apresentados. De acordo com os autores, esse modelo pode ser adaptado para programas que utilizam o paradigma de passagem de mensagem.

Os trabalhos descritos nessa seção concentram-se, essencialmente, em programas paralelos desenvolvidos em linguagens de programação concorrentes que usam o paradigma de programação paralela por memória compartilhada. Quando são considerados programas paralelos no paradigma de passagem de mensagem, poucos são os trabalhos sobre critérios de teste. A próxima seção aborda critérios estruturais de teste para o paradigma de passagem de mensagem que estão mais relacionados com este trabalho.

4.4 Teste no Paradigma de Passagem de Mensagem

A proposta de Yang et al. [65] para teste de programas concorrentes em Ada foi estendida por Vergilio et al. [60], que propõem um modelo de teste e critérios estruturais de teste, baseados em fluxo de controle e fluxo de dados, para ambiente de passagem de mensagem. Esse modelo considera que um número n fixo e conhecido de processos paralelos é criado durante a inicialização do programa paralelo. O programa paralelo $Prog$ é formado por um conjunto de n processos, $Prog = \{P^0, P^1, \ldots, P^{n-1}\}$. Cada processo é representado por um Grafo de Fluxo de Controle (GFC^p, tal que $0 \leq p \leq n-1$), que é construído utilizando-se dos mesmos conceitos utilizados na construção de um GFC seqüencial. Cada nó corresponde a um único comando e uma aresta corresponde à ligação entre os comandos e pode representar um desvio de fluxo de controle. Um nó pode ou não estar associado a um comando de comunicação.

Para o programa paralelo $Prog$, constrói-se um Grafo de Fluxo de Controle Paralelo (GFCP), composto dos GFC^p, que inclui a representação do fluxo de comunicação entre os processos. No GFC^p, o conjunto de nós N é dado por n_i^p, nó i do processo p. Definem-se os conjuntos Ns e Nr, subconjuntos de N, como sendo compostos pelos n_i^p que estejam associados a um comando de envio e recebimento de mensagem respectivamente.

Para cada nó $n_i^p \in Ns$ associa-se um conjunto de nós $R_i^p \subseteq Nr$, que representa os possíveis receptores da mensagem enviada por n_i^p.

O GFCP possui dois tipos de aresta:

- Arestas intra-processo:
  $E_i^p = \{(n_j^p, n_k^p) \mid 0 \leq p \leq n-1\}$
- Arestas inter-processos (ou de comunicação):
  $Es = \{(n_j^a, n_k^b) \mid n_j^a \in Ns,\ n_k^b \in R_j^a\}$

4.4.1 Exemplo de GFCP

A Figura 4.1 apresenta o GFCP para o código do exemplo da Listagem 3.1, considerando dois processos. As arestas Es são representadas por arestas tracejadas no grafo, nas quais a origem está no nó de envio e o destino está no nó de recepção da mensagem.

Seguem os conjuntos definidos para o exemplo:
$n = 2$

$Prog = \{P^0, P^1\}$

$N = \{1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 11^0, 1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 10^1, 11^1\}$

$Ns = \{6^0, 6^1\}$

$Nr = \{8^0, 8^1\}$

$R_6^0 = \{8^1\}$

$R_6^1 = \{8^0\}$

$E_i^0 = \{(1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (5^0, 6^0), (6^0, 7^0), (7^0, 11^0), (4^0, 8^0), (8^0, 9^0), (9^0, 10^0), (10^0, 11^0)\}$

$E_i^1 = \{(1^1, 2^1), (2^1, 3^1), (3^1, 4^1), (4^1, 5^1), (5^1, 6^1), (6^1, 7^1), (7^1, 11^1), (4^1, 8^1), (8^1, 9^1), (9^1, 10^1), (10^1, 11^1)\}$

$Es = \{(6^0, 8^1), (6^1, 8^0)\}$

$E = Es \cup E_i^p, 0 \leq p \leq n-1$

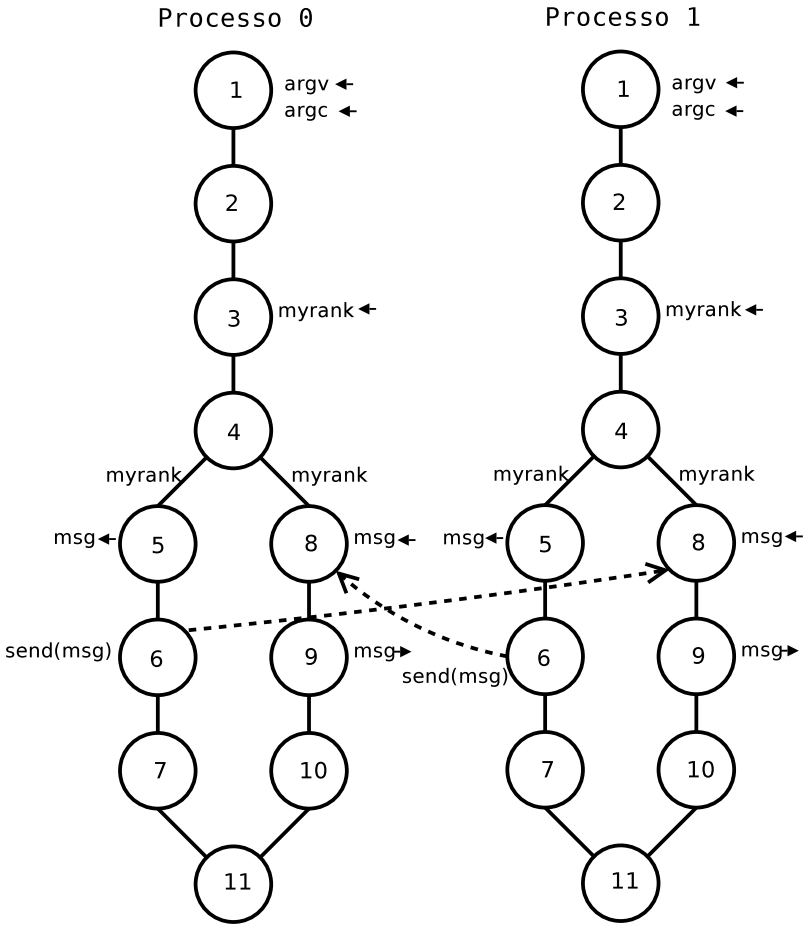


Figura 4.1: Exemplo de GFCP para o programa da Listagem 3.1, supondo dois processos paralelos

4.4.2 Caminhos e Associações

Os seguintes conceitos são necessários para o entendimento dos requisitos dos critérios usados.

Um caminho C^p no GFC^p, chamado de intra-processo, é dado por uma seqüência finita de nós, $C^p = (n_1, n_2, \ldots, n_m)$, tal que todos os nós pertencem ao mesmo processo p.

Um caminho C no GFCP, chamado de inter-processos, é dado pelo conjunto de caminhos intra-processos executados em paralelo e seqüências de sincronização. Uma seqüência de sincronizações é dada por uma seqüência finita de arestas inter-processos, $S^p = ((n_1^a, m_1), (n_2^b, m_2), \ldots, (n_j^l, m_j))$, tal que todos os nós m_j pertencem ao mesmo processo p. Portanto, $C = (C^0, C^1, \ldots, C^{n-1}, S^0, S^1, \ldots, S^{n-1})$.

Um caminho $C^p = (n_1, n_2, \ldots, n_m)$ é dito ser simples se todos os seus nós são distintos, exceto possivelmente o primeiro e o último. Ele é livre de laço se todos os seus nós são distintos. Um caminho é dito ser completo se n_1 e n_m são os nós iniciais e finais do processo p, respectivamente.

Um caminho $C = (C^0, C^1, \ldots, C^{n-1}, S^0, S^1, \ldots, S^{n-1})$, será simples se todos os C^i forem simples. Será livre de laço se todos os C^i forem livre de laço. Será completo se todos os C^i forem completos.

Um caminho C inclui um nó n_i^p (ou uma aresta $(n_j^p, n_k^p) \in E_i^p$) se incluir um caminho intra-processo C^p que inclui o nó (ou aresta) dado. Um caminho C inclui a aresta $(n_j^a, n_k^b) \in Es$ se incluir uma seqüência de sincronizações S^b que inclui a aresta inter-processos dada.

Um caminho é livre de definição c.r.a (com relação a) x de n_j para n_k, e de n_j para a aresta (n_m, n_k) se $x \in def(n_1^p)$ e $x \notin def(n_i^p)$, para $j < i \leq k$. O conjunto $def(n_i^p) = \{\text{variáveis } x \mid x \text{ é definida em } n_i^p\}$.

Um uso de uma variável x ocorre sempre que existir uma referência ao valor de x, armazenado na posição de memória. Um uso pode ser diferenciado em três tipos: 1) c-uso, análogo ao c-uso seqüencial; 2) p-uso, análogo ao p-uso seqüencial; e 3) s-uso (ou uso comunicacional), um comando de envio de mensagem, associado a uma aresta Es. Desta maneira, uma associação definição uso pode ser:

Associação c-uso: a tripla $\langle n_i^p, n_j^p, x\rangle \mid x \in def(n_i^p)$, n_j^p possui um c-uso de x e existe um caminho livre de definição c.r.a x de n_i^p para n_j^p.

Associação p-uso: a tripla $\langle n_i^p, (n_j^p, n_k^p), x\rangle \mid x \in def(n_i^p)$, (n_j^p, n_k^p) possui um p-uso de x e existe um caminho livre de definição c.r.a x de n_i para (n_j^p, n_k^p).

Associação s-uso: a tripla $\langle n_i^a, (n_j^a, m_k^b), x\rangle \mid x \in def(n_i^a)$, (n_j^a, m_k^b) possui um s-uso de x e existe um caminho livre de definição c.r.a x de n_i^a para (n_j^a, m_k^b).

Considerando as associações s-uso e a comunicação entre processos, outras associações inter-processos podem ser definidas:

Associação s-c-uso: a quádrupla $\langle n_i^a, m_l^b, x^a, y^b\rangle \mid$ existe uma associação do tipo s-uso $\langle n_i^a, (n_j^a, m_k^b), x^a\rangle$ e uma associação do tipo c-uso $\langle m_k^b, m_l^b, y^b\rangle$.

Associação s-p-uso: a quádrupla $\langle n_i^a, (m_l^b, m_m^b), x^a, y^b\rangle \mid$ existe uma associação do tipo s-uso $\langle n_i^a, (n_j^a, m_k^b), x^a\rangle$ e uma associação do tipo p-uso $\langle m_k^b, (m_l^b, m_m^b), y^b\rangle$.

Essas associações permitem verificar a existência de erros após a comunicação entre processos, durante o uso da informação recebida pela comunicação. Com base nesses conceitos, um conjunto de critérios de teste são apresentados a seguir.

4.4.3 Critérios Estruturais

Critérios baseados em fluxo de controle e no fluxo de comunicação:

Todos-nós-s: requer o exercício de todos os nós do conjunto Ns.

Todos-nós-r: requer o exercício de todos os nós do conjunto Nr.

Todos-nós: requer o exercício de todos os nós do conjunto N.

Todas-arestas-s: requer o exercício de todas as arestas do conjunto Es.

Todas-arestas: requer o exercício de todas as arestas do conjunto E.

Critérios baseados em fluxo de dados e em passagem de mensagem:

Todas-defs: para cada nó n_i^p e cada x em $def(n_i^p)$, uma associação definição uso c.r.a x, de qualquer tipo é exercitada.

Todas-defs/s: para cada nó n_i^p e cada x em $def(n_i^p)$, uma associação definição s-c-uso ou s-p-uso c.r.a x é exercitada, ou uma definição de qualquer outro tipo se um uso inter-processos de x não existir.

Todos-c-usos: requer o exercício de todas as associações c-uso.

Todos-p-usos: requer o exercício de todas as associações p-uso.

Todos-s-usos: requer o exercício de todas as associações s-uso.

Todos-s-c-usos: requer o exercício de todas as associações s-c-uso.

Todos-s-p-usos: requer o exercício de todas as associações s-p-uso.

4.4.4 Exemplo de Aplicação dos Critérios

Aplicando-se os critérios de teste da subseção anterior, no exemplo da Listagem 3.1 (Figura 4.1), obtemos os seguintes conjuntos:

Associações c-uso:

$\langle 8^0, 9^0, msg \rangle$
$\langle 8^1, 9^1, msg \rangle$

Associações p-uso:

$\langle 3^0, (4^0, 5^0), myrank \rangle$
$\langle 3^0, (4^0, 8^0), myrank \rangle$
$\langle 3^1, (4^1, 5^1), myrank \rangle$
$\langle 3^1, (4^1, 8^1), myrank \rangle$

Associações s-uso:

$\langle 5^0, (6^0, 8^1), msg \rangle$
$\langle 5^1(6^1, 8^0), msg \rangle$

Associações s-c-uso:

$\langle 5^0, 9^1, msg^0, msg^1 \rangle$
$\langle 5^1, 9^0, msg^1, msg^0 \rangle$

Note que os elementos requeridos abaixo são não executáveis pois, não existe dado de teste, entradas e sincronizações, que causem a execução de caminhos que cubram esses elementos requeridos.
$\langle 8^0, 9^0, msg \rangle$
$\langle 3^0, (4^0, 5^0), myrank \rangle$
$\langle 3^1, (4^1, 8^1), myrank \rangle$
$\langle 5^1(6^1, 8^0), msg \rangle$
$\langle 5^1, 9^0, msg^1, msg^0 \rangle$

4.5 Determinismo

Como visto anteriormente, o não-determinismo é um fator que pode estar presente nos programas paralelos, o que representa uma dificuldade adicional para testá-los. Conseguir testar deterministicamente é importante por dois motivos: 1) se um erro for encontrado será necessário corrigir o programa e reproduzir o caminho executado anteriormente para verificar se o erro foi efetivamente corrigido e novos erros não foram inseridos e 2) o usuário deve ter um mecanismo para cobrir sincronizações executáveis sem a necessidade de executar o programa um número indefinido de vezes. Existem algumas abordagens para obter-se testes determinísticos: a execução controlada, o teste temporal (*timing-related testing*) e execução controlada não-intrusiva.

A execução controlada em programas concorrentes da linguagem Ada é tratada por Tai et al. [50] e Carver e Tai [8] através da instrumentação do programa. Os dados de sincronização são coletados durante a execução do programa, esses dados são utilizados por um mecanismo de controle que adiciona semáforos e monitores que forçam a reprodução do teste.

O teste temporal [63] consiste na inserção de *delays* ou na execução do programa múltiplas vezes para tentar diferentes sincronizações. Long et al. [32] usam teste temporal com monitores para a linguagem Java. Um monitor protege os dados com os quais ele opera e sincroniza as chamadas de procedimento. Usando um relógio externo para sincronizar as chamadas ao monitor pode-se controlar a interação do processo de testes sem alterar o código testado, garantindo o exercício das pré-condições desejadas.

Na execução controlada não-intrusiva [12] um mecanismo de controle coleta informações sobre envio e recebimento de mensagens em PVM, esse mecanismo é implementado em programa que executa juntamente com cada processo paralelo. O programa paralelo é representado por um grafo de tarefas, quando usado para testes, a execução controlada explora todos os possíveis cenários de condições de corrida. A complexidade do algoritmo de teste depende do número de mensagens que são aceitáveis em um *receive*.

4.6 Ferramentas de Teste para Programas Paralelos

A maioria das ferramentas de teste disponíveis para a programação paralela auxiliam na análise de desempenho, visualização, monitoramento e depuração. Alguns exemplos destas ferrametas são: TDC Ada [50], ConAn (*Concurrency Analyser*) [32], Xab [3],

MDB [12], Paragraph [24], Della Pasta (*DELaware PArallel Software Testing Aid*) [65], STEPS (*Structural TESting of Parallel Software*) [33, 40], Astral (Ambiente de Simulação e Teste de pRogramas parALelos) [38], Umpire [61], Visit (*Visualize it!*) [27], XPVM [28] e XMPI [56]. Essas ferramentas são apresentadas a seguir. A Tabela 4.1 compara suas principais características.

TDC Ada apóia a depuração de programas na linguagem Ada, utilizando a execução determinística.

ConAn gera *drivers* para o teste de unidade em classes de programas concorrentes na linguagem Java, utilizando um relógio externo para a execução determinística.

Della Pasta foi uma das primeiras ferramentas de teste com análise de caminhos em programas de memória compartilhada. Um analisador estático gera os possíveis caminhos para cobrir associações entre as variáveis envolvidas na sincronização de tarefas.

Xab e XPVM permitem o monitoramento de eventos em programas PVM.

Visit permite a depuração e análise de desempenho de programas PVM.

MDB permite a execução controlada e a depuração de programas PVM.

Paragraph permite o monitoramento de programas MPI.

XMPI permite a visualização e monitoramento durante, ou após, a execução de programas da implementação LAM/MPI [57].

Umpire permite a depuração e monitoramento de programas MPI.

STEPS permite o monitoramento através da análise simbólica de cenários de teste e execução controlada em programas PVM, além de suportar um critério de teste estrutural baseado em fluxo de controle com base nas sincronizações, semelhante ao critério todas-arestas-s [60].

Astral permite o monitoramento da simulação de execução, feita pela análise simbólica do programa PVM. São implementados critérios baseados em fluxo de controle. A ferramenta supõe que o programa é determinístico. É possível, ainda, identificar problemas relacionados ao desempenho.

Tabela 4.1: Comparativo entre as ferramentas

Ferramenta	Ambiente ou linguagem	Critérios estruturais	Determinismo da execução	Monitoramento ou visualização	Análise de desempenho
TDC Ada	Ada		√		
ConAn	Java		√		
Della Pasta	Mem.comp.	√	√	√	
Xab	PVM			√	
XPVM	PVM			√	
Visit	PVM			√	√
MDB	PVM		√	√	
STEPS	PVM	√	√	√	
Astral	PVM	√		√	√
Paragraph	MPI			√	√
XMPI	MPI			√	
Umpire	MPI			√	

A maioria das ferramentas acima monitoram eventos gerados pelo programa em tempo de execução ou fazem a análise simbólica estática dos programas, possibilitando assim um meio de depuração. Uma revisão das ferramentas de teste de desempenho para programas em MPI é apresentada por Browne et al. [5]. Entretanto, nenhuma dessas ferramentas dá suporte à aplicação de critérios estruturais baseados em fluxo de dados e fluxo de controle.

4.7 Considerações Finais

Os critérios existentes para teste de programas paralelos concentram-se, principalmente, no paradigma de memória compartilhada. Não existe ferramenta de teste, que apóie o uso de critérios estruturais baseados em fluxo de dados, para programas paralelos que sigam o paradigma de passagem de mensagem, MPI em particular.

Critérios específicos para o ambiente de passagem de mensagem, introduzidos na Seção 4.4, foram propostos no contexto do projeto ValiPVM. No mesmo contexto foi proposta uma arquitetura de ferramenta, chamada ValiPar [47], para auxiliar a aplicação desses critérios. A ferramenta ValiPar é multi linguagem e permite sua configuração para diversos ambientes de passagem de mensagem. Está sendo atualmente configurada para PVM (ValiPVM). O próximo capítulo apresenta a ferramenta ValiMPI, que apóia a aplicação de tais critérios, além de permitir a execução controlada e depuração. A ValiMPI corresponde à implementação da especificação da ValiPar para o ambiente MPI.

5 A Ferramenta ValiMPI

Este capítulo trata da arquitetura, implementação e uso da ferramenta de teste ValiMPI. Essa ferramenta corresponde à instanciação da arquitetura proposta pela ValiPar [47] para MPI. O objetivo da ferramenta ValiMPI é a verificação e validação de programas paralelos em MPI através da aplicação dos critérios estruturais específicos para ambiente de passagem de mensagem descritos no capítulo anterior.

Os programas testados pela ferramenta ValiMPI devem ser escritos em MPI na linguagem C, todavia essa restrição pode ser transposta através de modificações nos módulos *Vali-Inst* e *Vali-Exec*, descritas nas considerações finais deste capítulo.

5.1 Considerações Iniciais

A ferramenta ValiMPI considera a presença dos seguintes itens para seu correto funcionamento:

- Ambiente UNIX e shell compatível com bash [59].
- Implementação do MPI 1.1 [36] ou superior.
- IDeL [44, 45], instanciada para a linguagem C.
- Programa alvo dos testes em MPI, sintaticamente correto na linguagem C.
- Ferramentas auxiliares [59]: sed, indent, wc e awk.
- Variável de ambiente VALIMPI_ROOT apontando para o diretório de instalação da ferramenta.

5.2 Arquitetura da Ferramenta

A ferramenta ValiMPI fornece funções para criar sessões de teste, gerar e executar casos de teste, avaliar a cobertura dos testes em relação aos seguintes critérios (descritos na Seção 4.4): todos-nós, todos-nós-r, todos-nós-s, todas-arestas, todas-arestas-s, todos-c-usos, todos-p-usos e todos-s-usos. Para realizar essas atividades, a ferramenta conta com quatro módulos principais que se comunicam através de arquivos, conforme ilustrado na Figura 5.1. O funcionamento de cada módulo e a interação entre os mesmos são detalhados nas subseções a seguir. Em cada subseção, o exemplo da Listagem 3.1, rodando dois processos paralelos conforme ilustrado pela Figura 4.1, é usado para elucidar o módulo em questão.

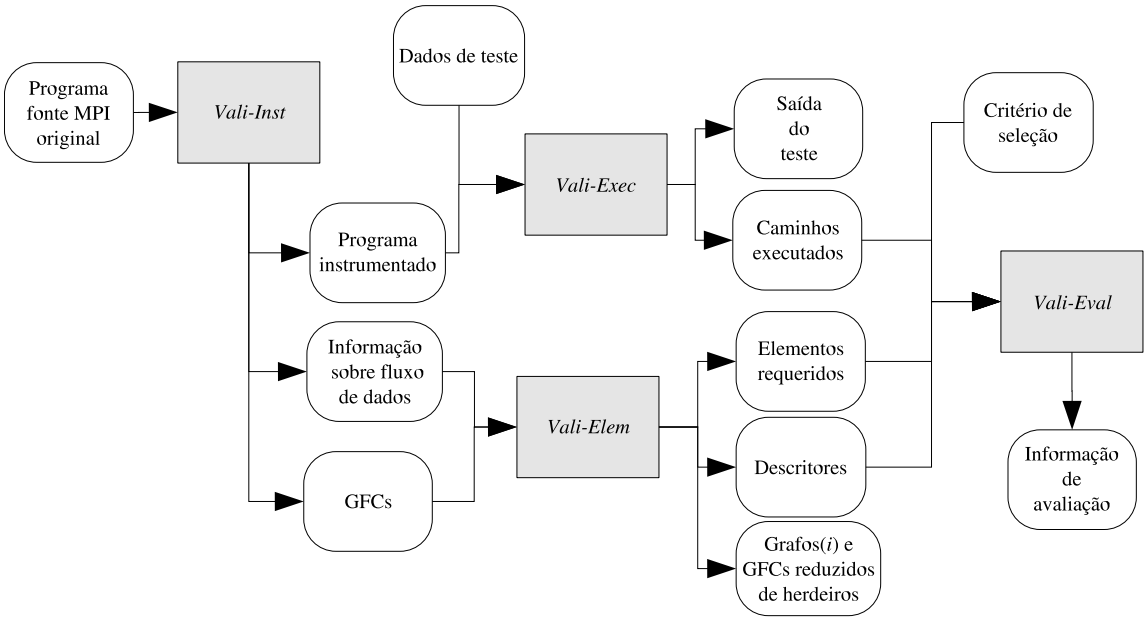


Figura 5.1: Arquitetura de módulos da ferramenta ValiMPI

5.2.1 Instrumentação e Extração das Informações de Fluxo

A tarefa de instrumentação e extração das informações de fluxo é realizada pelo módulo executável *Vali-Inst* (Figura 5.2). Esse módulo recebe como entrada o programa fonte original e a descrição semântica da instrumentação e gera em sua saída o programa fonte instrumentado, os grafos de fluxo de controle e as informações sobre fluxo de dados.

A tarefa de instrumentação propriamente dita é delegada à IDeL [45]. A IDeL, instanciada para a gramática da linguagem C, recebe o programa fonte MPI juntamente com a descrição semântica da instrumentação. Essa descrição semântica é o que viabi-

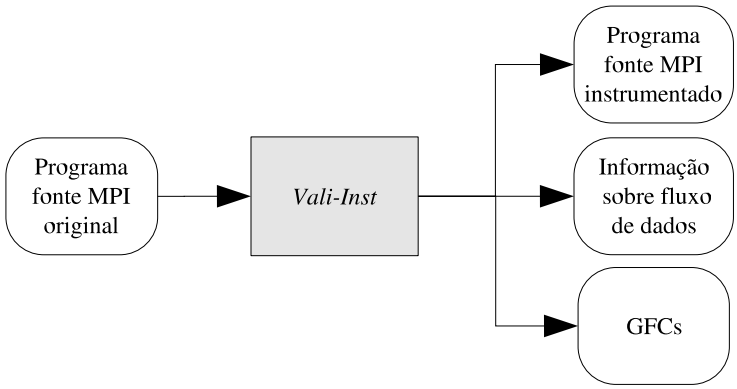


Figura 5.2: Módulo *Vali-Inst*

liza a instrumentação, pois de acordo com um padrão gramatical casado, uma ação de instrumentação é tomada.

O programa fonte original é processado pela IDeL e um programa fonte instrumentado é gerado de acordo com a descrição semântica passada. O programa instrumentado possui comandos tipo *check-point* para identificação do traço de execução. Também são gerados pelo módulo *Vali-Inst*: os grafos de fluxo, contendo informações sobre desvios, definição e uso de variáveis, além de informações sobre a troca de mensagens.

O *Vali-Inst* substitui as chamadas de funções MPI_Send por ValiMPI_Send, MPI_Isend por ValiMPI_Isend, MPI_Recv por ValiMPI_Recv e MPI_Irecv por ValiMPI_Irecv, que são versões das funções originais que possibilitam a identificação das sincronizações e a execução controlada. As demais funções de comunicação ponto-a-ponto da Tabela 3.2 não foram implementadas por serem apenas especializações das funções MPI_Send e MPI_Isend.

As chamadas de função de teste e espera de requisições de envio ou recebimento de mensagens não-bloqueantes também são substituídas, ValiMPI_Test substitui MPI_Test enquanto ValiMPI_Wait substitui MPI_Wait. Essa mudança permite a geração correta dos traços de execução sem alterar a semântica do programa em teste, identificação das sincronizações e a execução controlada.

A escolha de substituir os nomes das chamadas de função implica na necessidade de que todas as unidades alvos de teste estejam nos arquivos fontes instrumentados, além disso, se uma das funções descritas acima for executada por sua versão original, não será possível garantir a execução controlada.

As funções de início e término do ambiente paralelo também têm suas versões modificadas implementadas, ValiMPI_Init e ValiMPI_Finalize respectivamente, essas funções são

responsáveis pela inicialização de variáveis e abertura e fechamento de arquivos. Todavia, o programa instrumentado mantém a chamada para os nomes originais das funções. A substituição é feita em tempo de "*link*-edição" do executável através do mecanismo de *profiling* descrito no padrão MPI 1.1. Essa escolha possibilita que as chamadas de início e término do ambiente paralelo possam estar presentes em arquivos fontes não instrumentados.

Finalmente, os arquivos temporários são apagados e o programa instrumentado é indentado para possibilitar a sua leitura por parte do usuário.

5.2.1.1 Exemplo de Uso

Executando-se o comando de instrumentação (Apêndice B) para a Listagem 3.1: vali_inst mpi_hello.c, obtém-se o programa fonte instrumentado mpi_hello.c_instrumentado.c, Listagem 5.1 e o grafo definição-uso, Listagem 5.2, correspondentes ao grafo da Figura 4.1.

A Listagem 5.1 corresponde ao arquivo fonte original acrescido dos *check-points* de geração do traço de execução e identificação das sincronizações. Na linha 1, onde havia a inclusão de mpi.h no programa original, agora há a inclusão de valimpi.h. No bloco de declarações de variáveis de cada função, há duas macros[1]: VALIMPI_TRACE_DECL e VALIMPI_REQ_LIST_DECL. A primeira, linha 7, é responsável pela declaração da variável valimpiTrace, que é corresponde à fila de nós armazenados no traço de execução da função no processo que a executa. Enquanto a segunda, linha 8, é responsável pela declaração da variável valimpiListReq, que é a lista de requisições não-bloqueantes pendentes da função no processo que a executa.

O traço de execução de cada função em relação ao processo que a executa é inicializado por ValiMPI_Init_trace, linha 10, que inicializa valimpiTrace com uma fila vazia e cria o arquivo ⟨nome da função⟩.p⟨número do processo⟩ no diretório de teste. Apesar de existirem *check-points* consecutivos com o mesmo número de nó, como nas linhas 18 e 19, a fila de traço não armazena entradas repetidas consecutivas, pois isso inviabilizaria a avaliação correta das arestas.

A lista de requisições de cada função em relação ao processo que a executa é inicializada por ValiMPI_Req_list_init, linha 9, que inicializa valimpiListReq com uma lista vazia. A lista de requisições é necessária para que a fila de traço de execução não escreva em arquivo enquanto houverem requisições pendentes geradas por *receives* não-bloqueantes,

[1]As declarações são macros porque a IDeL restringe declarações de tipos não primitivos no arquivo de descrição semântica mpi.idel.

o que acarretaria em "furos" no traço de execução.

Uma sincronização (n_i^a, m_j^b) aparece no traço de execução do processo emissor da mensagem, P^a, da seguinte forma: $n_{i-1}^a\ n_i^a\ 0^b\ n_i^a\ n_{i+1}^a$. E no processo receptor da mensagem, P^b, da seguinte forma: $m_{j-1}^b\ m_j^b\ n_i^a\ m_j^b\ m_{j+1}^b$. O nó 0^b aparece no traço de P^a para identificar o processo destino do *send*, o nó receptor não está presente no traço de P^a pois isto implicaria em uma sincronização adicional, o que complicaria o caso de *sends* não-bloqueantes. O nó n_i^a aparece no traço de P^b como nó receptor, pois este dado é empacotado junto adicionalmente no processo emissor. Um exemplo efetivo de traço de execução é mostrado nas Listagens 5.5 e 5.6.

A função de *check-point* Valimpi_Check_trace (Apêndice A) corresponde à inserção do nó na fila, se o mesmo não for o último elemento da fila, e posterior escrita no arquivo de traço de execução, enquanto a lista de requisições estiver vazia. A função de *check-point* Valimpi_Send_trace, linha 23, além de inserir na fila de traço o processo destino, nó 0^b, envia o nó do *send*, n_i^a, juntamente com a mensagem. A função de *check-point* Valimpi_Recv_trace, linha 28, recebe o nó do *send* e insere-o na fila de traço, posteriormente o próprio nó do *receive*, m_j^b, é inserido na fila. A função de *check-point* do *receive* também é responsável pela manutenção da fila de seqüências de sincronização, representada pelo arquivo seq_sync.p⟨número do processo⟩. Os procedimentos de *check-point* para mensagens não-bloqueantes são análogos. Os funcionamento dos procedimentos de *check-point* para a execução controlada são descritos na Subseção 5.2.3.1.

Listagem 5.1: Arquivo fonte instrumentado

```
#include "valimpi.h"
#include "mpi_hello.h"
int main(int argc, char *argv[])
{
    char msg[10];
    int myrank;
    VALIMPI_TRACE_DECL;
    VALIMPI_REQ_LIST_DECL;
    ValiMPI_Req_list_init(&valimpiListReq);
    ValiMPI_Init_trace(&valimpiTrace, "main");
    ValiMPI_Check_trace(&valimpiTrace, 1); { {
    ValiMPI_Check_trace(&valimpiTrace, 2);
    MPI_Init(&argc, &argv); } {
    ValiMPI_Check_trace(&valimpiTrace, 3);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); } {
    ValiMPI_Check_trace(&valimpiTrace, 4);
```

```
    if (myrank == 0) {
        ValiMPI_Check_trace(&valimpiTrace, 5); { { {
        ValiMPI_Check_trace(&valimpiTrace, 5);
20      strcpy(msg, "Hello!"); } { {
        ValiMPI_Check_trace(&valimpiTrace, 6);
        ValiMPI_Send_trace(&valimpiTrace, msg, strlen(msg) + 1, MPI_CHAR,
            1, 99, MPI_COMM_WORLD); } } }
        ValiMPI_Check_trace(&valimpiTrace, 7); }
    }
25  else {
        ValiMPI_Check_trace(&valimpiTrace, 8); { { { {
        ValiMPI_Check_trace(&valimpiTrace, 8);
        ValiMPI_Recv_trace(&valimpiTrace, msg, sizeof(msg), MPI_CHAR, 0,
            99, MPI_COMM_WORLD, &status);  } } {
        ValiMPI_Check_trace(&valimpiTrace, 9);
30      printf("received: %s\n", msg); } }
        ValiMPI_Check_trace(&valimpiTrace, 10); }
    } } {
    ValiMPI_Check_trace(&valimpiTrace, 11);
    MPI_Finalize(); } {
35  ValiMPI_Check_trace(&valimpiTrace, 11); {
    ValiMPI_End_trace();
    return 0; } } }
    ValiMPI_End_trace();
    ValiMPI_Check_trace(&valimpiTrace, 11);
40  }
```

Um arquivo com informações de fluxo de dados e o fluxo de controle é gerado para cada função do arquivo instrumentado. A Listagem 5.2 ilustra o arquivo gerado correspondente ao arquivo da Listagem 3.1. Este arquivo apresenta as definições, linha 3; c-usos, linha 25; p-usos, linha 12; s-usos, linha 6; e definições por dereferenciação, ou seja, possíveis definições na passagem de argumentos usando a notação de endereço "&" da linguagem C. Os *sends* e *receives* são marcados por *m_send* e *m_recv*, respectivamente, seguidos dos seus argumentos. Abaixo das informações de fluxo, linha 26, segue a representação textual do GFC.

Listagem 5.2: GFC e grafo definição-uso da função main

```
digraph gfc {
node [shape = circle] 1;
/* definition of argc at 1 */
/* definition of argv at 1 */
```

```
node [shape = doublecircle] 11;
node [shape = circle] 2;
/* derefdefinition of argc at 2 */
/* derefdefinition of argv at 2 */
node [shape = circle] 3;
/* derefdefinition of myrank at 3 */
node [shape = circle] 4;
/* pusage of myrank at 4 */
node [shape = circle] 7;
node [shape = circle] 5;
/* cusage of msg at 5 */
node [shape = circle] 6;
/* m_send of msg,strlen(msg)+1,MPI_CHAR,1,99,MPI_COMM_WORLD at 6 */
/* susage of msg at 6 */
node [shape = circle] 10;
node [shape = circle] 8;
/* m_recv of msg,sizeof(msg),MPI_CHAR,0,99,MPI_COMM_WORLD,&status at 8 */
/* definition of msg at 8 */
/* derefdefinition of status at 8 */
node [shape = circle] 9;
/* cusage of msg at 9 */
1 -> 2;
2 -> 3;
3 -> 4;
4 -> 5;
4 -> 8;
7 -> 11;
10 -> 11;
5 -> 6;
6 -> 7;
8 -> 9;
9 -> 10;
 }
```

5.2.1.2 Limitações do Módulo

A IDeL, tal qual instanciada para a gramática da linguagem C, apresentou alguns problemas com relação aos comentários da linguagem C e a presença de linhas em branco, para contornar esse problema, o módulo *Vali-Inst* suprime comentários e linhas em branco. A presença de tipos não primitivos (declarados com typedef) abortam a IDeL, para contornar esse problema esses tipos, quando necessários, são declarados em outro arquivo fonte,

que não é instrumentado. A IDeL não gera o arquivo do grafo se a função instrumentada usar tipo de retorno implícito, e.g. main ao invés de int main ou void main.

Quando há uma declaração de variável em um escopo menor que o escopo da função, a IDeL conta os nós erradamente. A IDeL também não identifica definição de variável quando esta ocorre junto à declaração. A correção desses problemas implicaria na reescrita dos arquivos da gramática da linguagem C que foram compilados pela IDeLgen para formar o instrumentador IDeL.C.

Por tratar apenas de gramáticas livres de contexto, a IDeL não identifica automaticamente definições de variável por dereferenciação, para o caso de variáveis do tipo vetor ou apontador. Como o padrão MPI define claramente os parâmetros de entrada e saída de cada função, é feito um pós-processamento das funções MPI instrumentadas identificando as variáveis nelas definidas.

Pelo mesmo motivo, um tratamento adicional é dado às informações de fluxo de dados. As macros do MPI, identificadas pela IDeL como uso de variável, são removidas no pós-processamento. Os s-usos também são, inicialmente, identificados como c-usos, sendo necessário encontrá-los e substituí-los no pós-processamento.

5.2.1.3 Geração do Executável

O programa fonte instrumentado, junto com os demais fontes, devem ser compilados, incluindo o diretório de cabeçalhos $VALIMPI_ROOT/include. O programa executável a ser testado deve ser ligado à biblioteca de *profiling* do MPI (libpmpi.a) e à biblioteca ValiMPI (libvalimpi.a) definida em $VALIMPI_ROOT/lib, para viabilizar a geração do traço de execução e a execução controlada dos casos de teste.

O *script* *Vali-cc* (Figura 5.3) auxilia o usuário na tarefa de compilação do aplicativo descrita no parágrafo anterior, incluindo automaticamente as bibliotecas e os cabeçalhos necessários.

Figura 5.3: Módulo *Vali-cc*

5.2.2 Geração dos Elementos Requeridos

A tarefa de geração dos elementos requeridos é realizada pelo módulo executável *Vali-Elem* (Figura 5.4) [58]. Esse módulo recebe como entrada os GFCs, as informações de fluxo de dados e a distribuição das funções em teste entre os processos, gerando em sua saída os elementos requeridos pelos critérios baseados em fluxo de controle e fluxo de dados. Também são gerados descritores dos elementos requeridos. Para isso são utilizados dois outros grafos[2]: o grafo reduzido de herdeiros e o grafo(i).

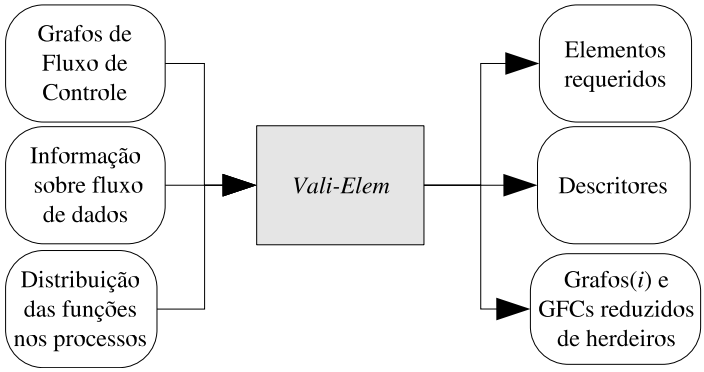


Figura 5.4: Módulo *Vali-Elem*

Em um grafo reduzido de herdeiros [11] todas as arestas são primitivas. O algoritmo de geração do grafo baseia-se no fato de existirem arestas dentro de um GFC que são sempre executadas quando outra aresta é executada. Se todo caminho completo que inclui a aresta a sempre incluir a aresta b, então b é chamada de aresta herdeira de a e a é chamado ancestral de b, pois b herda informação de execução de a, ou seja, uma aresta que sempre é executada quando se executa uma outra aresta é denotada por aresta herdeira.

O conceito de aresta primitiva é, então, estabelecido em função do conceito de aresta herdeira, sendo entendido como aresta primitiva toda aresta que não é herdeira de nenhuma outra. Ao conceito de aresta primitiva foi acrescentado o conceito de aresta de comunicação, que é uma aresta que liga dois processos em um GFCP. Utilizando o conceito de aresta primitiva e de aresta de comunicação é possível minimizar o número de arestas requeridas pela ValiMPI.

Além dos grafos reduzidos de herdeiros, um grafo(i) é construído para cada nó i que contenha definição de variável, sendo que, um dado nó k pertencerá a um grafo(i), se existir, pelo menos um caminho de i para k livre de definição com relação a pelo menos uma variável V_1 definida em i. Considerando $C(i, k)$ o conjunto de todos os possíveis

[2]Também utilizados na ferramenta *PokeTool* [9].

caminhos do nó i ao nó k, no GFC, então k pertencerá a um grafo(i) e por conseguinte, todos os demais nós de um caminho C_1, se existir $C_1 \in C(i,k)$ e $V_1 \in def$(i), tal que C_1 é livre de definição c.r.a V_1.

Portanto, um mesmo nó, ou aresta, do grafo pode dar origem a vários nós, ou arestas, no grafo(i), pois é construído apenas um grafo(i) para todas as variáveis definidas em i. Dessa forma, um nó k pode gerar diversas imagens distintas no grafo(i), visto que este é uma árvore, e portanto, todos os seus nós são distintos, pois não contém laços. Assim, para evitar caminhos intermináveis, causados pela existência de laços no GFC, em um mesmo caminho do grafo(i), apenas um nó pode conter mais de uma imagem, e sua imagem é o último nó do caminho. Desse modo, ao haver a aparição de uma segunda imagem k_2 de um nó k do GFC, num mesmo caminho do grafo(i), esse caminho é interrompido e seu último nó é exatamente k_2.

O grafo(i) é utilizado para se estabelecer associações definições e usos de variáveis, elementos requeridos por critérios baseados em fluxo de dados. Para cada elemento requerido, o *Vali-Elem* gera um descritor que será utilizado na avaliação (módulo *Vali-Eval*). Um descritor é uma expressão regular que descreve um caminho que cobre um elemento requerido. Por exemplo, os descritores para o critério todos-nós possuem o mesmo formato. Dado o nó $ni-p$ (notação textual para n_i^p) requerido, ele será coberto se em p um caminho que incluir n_i for executado. As expressões regulares que descrevem os descritores são apresentadas abaixo:

todos-nós $N^*\ ni-p\ N^*$, onde N é o conjunto de nós do GFC^p.

todos-nós-r $N^*\ ni-p\ N^*$, onde N é o conjunto de nós do GFC^p.

todos-nós-s $N^*\ ni-p\ N^*$, onde N é o conjunto de nós do GFC^p.

todas-arestas-s $N^*\ ni-p^a\ nj-p^b\ N^*$, onde N é conjunto de nós do GFC^b.

todas-arestas $N^*\ ni-p\ nj-p\ N*$, onde N é o conjunto de nós do GFC^p, OU $N^*\ ni-p^a\ nj-p^b\ N^*$, onde N é conjunto de nós do GFC^b.

todos-c-usos $N^*\ ni-p\ Nnv^*\ nj-p$, onde Nnv é o conjunto de nós do GFC^p que não redefinem x.

todos-p-usos $N^*\ ni-p\ Nnv^*\ nj-p\ [Nnv^*\ nj-p]^*\ nk-p$, onde Nnv é o conjunto de nós do GFC^p que não redefinem x.

todos-s-usos $Np^{a*}\ ni-p^a\ Np^a nv^*\ nj-p^a\ [Np^a nv^*\ nj-p^a]^*$ E $Np^{b*}\ ni-p^a\ nj-p^b\ Np^{b*}$, onde Np^a é conjunto de nós do GFC^a e, $Np^a nv$ é o conjunto de nós do GFC^a que não redefine x. Np^b é conjunto de nós do GFC^b.

5.2.2.1 Exemplo de Uso

Executando-se o gerador de elementos requeridos para dois processos da Listagem 3.1, a função main executa nos dois processos, vali_elem 2 "main(0,1)", obtém-se os elementos requeridos e descritores. A Listagem 5.3 corresponde aos elementos requeridos para o critério todos-nós e a Listagem 5.4 corresponde aos descritores desse mesmo critério.

Listagem 5.3: Elementos requeridos para o critério todos-nós

```
ELEMENTOS REQUERIDOS PARA O CRITERIO TODOS OS NOS

1) 1-0
2) 2-0
3) 3-0
4) 4-0
5) 5-0
6) 6-0
7) 7-0
8) 8-0
9) 9-0
10) 10-0
11) 11-0
12) 1-1
13) 2-1
14) 3-1
15) 4-1
16) 5-1
17) 6-1
18) 7-1
19) 8-1
20) 9-1
21) 10-1
22) 11-1
```

O descritor para o elemento requerido para o critério todos-nós: "1) 1-0" é representado, na primeira linha da Listagem 5.4, por um autômato (Figura 5.5) que é lido da seguinte maneira:

- Descritor número 1
- 1 autômato com 2 estados, estado final = 2.
- Estado 1
  - Transição para o estado 1: qualquer $n \in N \mid n \neq n_1^0$
  - Transição para o estado 2: n_1^0
  - Fim do estado
- Estado 2
  - Transição para o estado 2: qualquer $n \in N$
  - Fim do estado
- Fim do autômato

Listagem 5.4: Descritores para o critério todos-nós

```
1) 1 2 2 1 1:N/1-0 2:1-0 0 2 2:N 0
2) 1 2 2 1 1:N/2-0 2:2-0 0 2 2:N 0
3) 1 2 2 1 1:N/3-0 2:3-0 0 2 2:N 0
4) 1 2 2 1 1:N/4-0 2:4-0 0 2 2:N 0
5) 1 2 2 1 1:N/5-0 2:5-0 0 2 2:N 0
6) 1 2 2 1 1:N/6-0 2:6-0 0 2 2:N 0
7) 1 2 2 1 1:N/7-0 2:7-0 0 2 2:N 0
8) 1 2 2 1 1:N/8-0 2:8-0 0 2 2:N 0
9) 1 2 2 1 1:N/9-0 2:9-0 0 2 2:N 0
10) 1 2 2 1 1:N/10-0 2:10-0 0 2 2:N 0
11) 1 2 2 1 1:N/11-0 2:11-0 0 2 2:N 0
12) 1 2 2 1 1:N/1-1 2:1-1 0 2 2:N 0
13) 1 2 2 1 1:N/2-1 2:2-1 0 2 2:N 0
14) 1 2 2 1 1:N/3-1 2:3-1 0 2 2:N 0
15) 1 2 2 1 1:N/4-1 2:4-1 0 2 2:N 0
16) 1 2 2 1 1:N/5-1 2:5-1 0 2 2:N 0
17) 1 2 2 1 1:N/6-1 2:6-1 0 2 2:N 0
18) 1 2 2 1 1:N/7-1 2:7-1 0 2 2:N 0
19) 1 2 2 1 1:N/8-1 2:8-1 0 2 2:N 0
20) 1 2 2 1 1:N/9-1 2:9-1 0 2 2:N 0
21) 1 2 2 1 1:N/10-1 2:10-1 0 2 2:N 0
22) 1 2 2 1 1:N/11-1 2:11-1 0 2 2:N 0
```

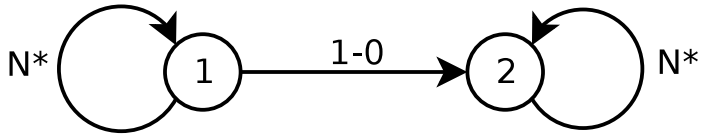


Figura 5.5: Autômato que reconhece o elemento requerido “1) 1-0”

5.2.3 Execução dos Casos de Teste

A tarefa de execução dos casos de teste é realizada pelo módulo *Vali-Exec* (Figura 5.6). Esse módulo recebe como entrada os dados de teste e o programa executável, gerando em sua saída o traço de execução e a seqüência de sincronizações de cada processo paralelo. Os dados de teste compreendem os argumentos de linha de comando, os dados entrados via teclado e o número de processos paralelos.

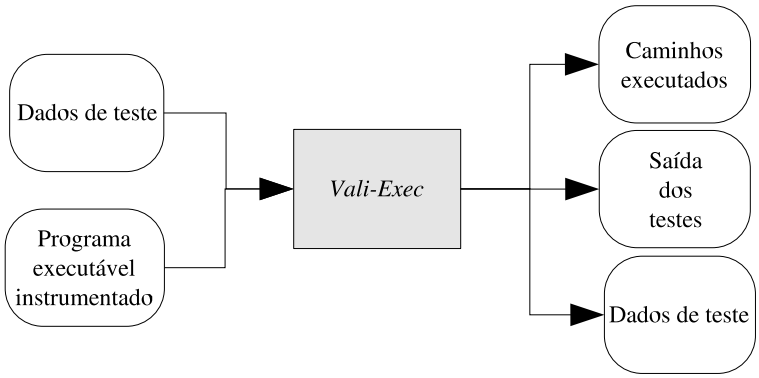


Figura 5.6: Módulo *Vali-Exec*

Esse módulo é implementado por um *script* que é responsável por iniciar o ambiente paralelo, se necessário, e executar o programa paralelamente.

O traço de execução é composto pelos caminhos inter-processos executados, para cada processo paralelo. O *Vali-Exec* armazena as entrada passadas (linha de comando e teclado), as saídas geradas, o número de processos paralelos, o traço de execução de cada função, por processo, e a seqüência de sincronizações em cada processo.

Durante a execução do programa, o usuário pode visualizar as saídas do *Vali-Exec* e do programa em teste para determinar se a saída obtida é igual a esperada. Caso não seja, um erro foi identificado e deve ser corrigido antes de prosseguir com o restante dos testes. O traço de execução também é gerado durante a execução do programa e pode servir de auxílio à depuração.

5.2.3.1 Execução Controlada

Ao usar este módulo, o usuário deve informar se o teste refere-se a um novo caso de teste ou à execução controlada. A execução controlada pode ser utilizada na reprodução de um caso de teste já executado, ou para tentar a cobertura de sincronizações não executadas. Assegurar que uma sincronização não executada é executável é responsabilidade do usuário.

São utilizadas dados de teste armazenados e a seqüência de sincronizações executadas do caso de teste que se deseja reproduzir. Ao final da execução, a saída produzida é comparada com a saída anterior, gerando um aviso ao usuário em caso de divergência.

A execução controlada do módulo *Vali-Exec* e implementada pela biblioteca valimpi, foi inspirada em Carver e Tai [50, 8]. A instrumentação permite que a coleta de dados de sincronização seja feita durante a execução do programa paralelo em teste. Esses dados, juntamente com as demais entradas do caso de teste, são utilizados na reexecução do teste para obter o comportamento determinístico da execução.

Cada processo enfileira as seqüências de sincronização executadas quando um *receive* é executado e opcionalmente o número de vezes em que a função de teste de recebimento de mensagem é chamada. Na execução controlada essas sincronizações, e o número de chamadas à função de teste de recebimento de mensagem se houver, são desenfileirados. O Apêndice A apresenta os pseudo-códigos das funções de *send* e *receive* que possibilitam a execução controlada.

5.2.3.2 Exemplo de Uso

A execução do caso de teste 1, em dois processos paralelos, é dada pela linha de comando vali_exec 1 run 2 mpi_hello_instrumentado. Essa execução gera os traços de execução para cada processo (Listagens 5.5 e 5.6) e as seqüências de sincronização de cada processo (Listagem 5.7).

Listagem 5.5: Traço de execução da função main no processo 0

1−0	2−0	3−0	4−0	5−0	6−0	0−1	6−0
	7−0	11−0					

Listagem 5.6: Traço de execução da função main no processo 1

1−1	2−1	3−1	4−1	8−1	6−0	8−1	9−1
	10−1	11−1					

Nos arquivos de seqüências de sincronização (seq_sync.p⟨número do processo⟩) cada linha representa o recebimento de uma mensagem na ordem ocorrida em tempo de execução. No exemplo da Listagem 5.7, a sincronização existente é lida: "o *receive* n_8^1 sincronizou com o *send* n_6^0, *tag* 99". Neste caso, como apenas o processo 1 tem um *receive*, o arquivo de seqüência de sincronizações do processo 0 está vazio. A abordagem de formar uma fila em um arquivo de seqüência por processo, foi utilizada para facilitar a alteração de sincronizações pelo usuário, basta alterar as linhas para tentar a execução das sincronizações desejadas.

Listagem 5.7: Seqüência de sincronização do processo 1

```
recv_no  8 :send_no 6 processo  0 com_a_tag 99
```

Para reexecutar esse mesmo caso de teste deterministicamente deve-se entrar a linha de comando vali_exec 1 rerun.

5.2.4 Avaliação da Cobertura

A tarefa da avaliação da cobertura dos casos de teste é realizada pelo módulo executável *Vali-Eval* (Figura 5.7) [58]. Esse módulo recebe como entrada um critério de teste selecionado pelo usuário, os descritores e elementos requeridos para o critério selecionado para os casos de teste executados pelo *Vali-Exec* e gera em sua saída informações sobre a cobertura do critério obtida pelos casos de teste.

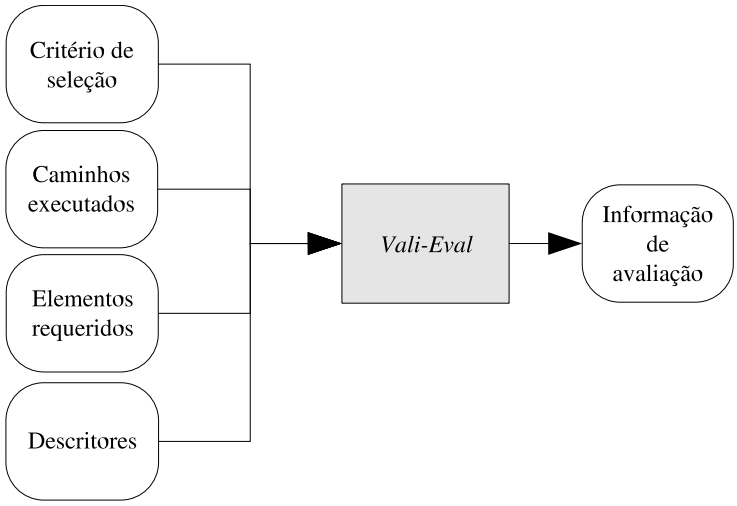


Figura 5.7: Módulo *Vali-Eval*

O *Vali-Eval* implementa um autômato que tem como entrada o caminho executado e verifica quais os descritores estão em seus estados finais, pois, isso significa que os

elementos requeridos descritos por aqueles descritores foram cobertos.

Os critérios de teste disponíveis são: todos-nós, todos-nós-s, todos-nós-r, todas-arestas, todas-arestas-s, todos-c-usos, todos-p-usos e todosss-usos, conforme definidos na Seção 4.4. Através do traço de execução dos casos de teste verificam-se quais os elementos requeridos pelo critério foram cobertos. É fornecido ao usuário o percentual de cobertura dos casos de teste, bem como a lista dos elementos requeridos não cobertos e a lista dos elementos requeridos cobertos acompanhada do caso de teste que os cobriram.

5.2.4.1 Exemplo de Uso

A avaliação dos casos de teste executados para o critério todos-nós, para o exemplo da Listagem 3.1, é dada pela linha de comando vali_eval 2 todos-nos "main(0,1)". O resultado da avaliação é dado pela Listagem 5.8. Nesse exemplo são avaliados dois processos paralelos, nos quais ambos correspondem a função main executando paralelamente nos processos 0 e 1.

Listagem 5.8: Avaliação do critério todos-nós

```
-- ELEMENTOS REQUERIDOS COBERTOS --
1) 1-0,  coberto por valimpi/test_case0001
2) 2-0,  coberto por valimpi/test_case0001
3) 3-0,  coberto por valimpi/test_case0001
4) 4-0,  coberto por valimpi/test_case0001
5) 5-0,  coberto por valimpi/test_case0001
6) 6-0,  coberto por valimpi/test_case0001
7) 7-0,  coberto por valimpi/test_case0001
11) 11-0,  coberto por valimpi/test_case0001
12) 1-1,  coberto por valimpi/test_case0001
13) 2-1,  coberto por valimpi/test_case0001
14) 3-1,  coberto por valimpi/test_case0001
15) 4-1,  coberto por valimpi/test_case0001
19) 8-1,  coberto por valimpi/test_case0001
20) 9-1,  coberto por valimpi/test_case0001
21) 10-1,  coberto por valimpi/test_case0001
22) 11-1,  coberto por valimpi/test_case0001

-- ELEMENTOS REQUERIDOS NÃO COBERTOS --
8) 8-0
9) 9-0
10) 10-0
16) 5-1
```

```
    17) 6-1
25  18) 7-1

Cobertura para o critério todos-nos: 72.73%
```

A Listagem 5.8 mostra duas listas, a primeira com os elementos requeridos cobertos e a segunda com os elementos requeridos que não foram cobertos. A lado de cada elemento requerido da primeira lista aparece um caso de teste que o cobriu. Ao final, um percentual de cobertura é informado ao usuário.

5.3 Considerações Finais

A implementação MPI utilizada neste trabalho foi a LAM/MPI versão 7.1.1 [57]. A utilização de outra implementação MPI requer o religamento da biblioteca ValiMPI e pode requerer a modificação do *script* de compilação *Vali-cc* e do módulo *Vali-Exec*, se a implementação não seguir o MPI 2.0, ou seja, não implementa o padrão de execução mpiexec.

Se a implementação MPI escolhida for a LAM/MPI, o módulo *Vali-Exec* gera o arquivo xmpi.lamtr no diretório do caso de teste. Esse arquivo pode ser interpretado pela ferramenta de apoio à depuração XMPI [56], Figura 3.3, compondo um auxílio adicional ao usuário.

Os grafos gerados pelo instrumentador podem ser visualizados pelo aplicativo dotty[3]. A visualização do grafo de fluxo pode auxiliar o usuário na geração de casos de teste para a cobertura dos elementos requeridos.

5.3.1 Adaptações para outras Linguagens e Ambientes

É possível adaptar a ferramenta *ValiMPI* para uso com outros ambientes de passagem de mensagem e linguagens de programação. Dois de seus módulos estão sujeitos a alterações: *Vali-Inst* e *Vali-Exec*, os outros módulos são independentes de linguagem e ambiente.

[3]http://www.graphviz.org

5.3.1.1 Adaptações para o Módulo *Vali-Inst*

Para o módulo *Vali-Inst*, modificar o ambiente de passagem de mensagem implicaria em:

1. Escrever um arquivo de descrição semântica ambiente.idel, análogo ao mpi.idel.
2. Adaptar o *script* vali-inst ao novo ambiente.
3. Reescrever a biblioteca valimpi, com as novas funções para implementar a geração dos arquivos de traço de execução, geração das seqüências de sincronização e execução controlada.

Mudar a linguagem de programação implicaria em:

1. Executar IDeLgen com a gramática da linguagem de programação desejada para criar uma nova IDeL.linguagem.
2. Modificar a descrição semântica do instrumentador (mpi.idel) para acomodar o casamento dos padrões desejados.

5.3.1.2 Adaptações para o Módulo *Vali-Exec*

Para o módulo *Vali-exec*, modificar o ambiente de passagem de mensagem implicaria em:

1. Modificar inicialização do ambiente paralelo.
2. A execução do programa paralelo propriamente dita, seguindo os parâmetros exigidos pelo módulo *Valimpi-Exec*.

A modificação de linguagem de programação, isoladamente, não implica em mudanças neste módulo.

6 Experimentos

Este capítulo apresenta os resultados de alguns experimentos do uso da ferramente ValiMPI. O objetivo destes experimentos é validar a ferramenta ValiMPI através do teste de programas paralelos em MPI. Os programas paralelos escolhidos para os experimentos representam diversas classes de problemas da programação paralela.

6.1 Materiais

O conjunto de Hardware e sistema operacional utilizado na condução dos experimentos é ilustrado pela Tabela 6.1. Os softwares utilizados e suas respectivas versões são ilustrados pela Tabela 6.2.

Tabela 6.1: Hardware e sistema operacional

Nome	Processador	Sistema operacional
andromeda	AMD Athlon XP 2000+	Debian GNU/Linux 2.6.5
aquarius	Intel Xeon 3.06GHz	Debian GNU/Linux 2.6.9
pyxis	Intel Pentium 4 2.4GHz	Debian GNU/Linux 2.4.26

Tabela 6.2: Software utilizado

Programa	Versão
LAM/MPI	7.1.1
XMPI	2.2.3b8
gcc	3.3.4
bash	2.05b.0
dotty	96c

6.2 Método

De posse do programa executável instrumentado, os passos abaixo são seguidos:

1. Geração de um conjunto inicial de dados de teste (T_{ini}). Este conjunto foi gerado manualmente para os experimentos PI, multiplicação de matrizes e produtor-consumidor e de aleatoriamente para os experimentos MDC e jantar dos filósofos.
2. Geração dos elementos requeridos para todos os critérios disponíveis pela *Vali-elem*.
3. Submissão do conjunto T_{ini} à *Vali-exec* e obtenção da cobertura inicial de cada critério pela *Vali-eval*.
4. Entrada manual de novos casos de teste ($T \mid T_{ini} \subseteq T$) e nova avaliação da cobertura até que todos os elementos requeridos considerados executáveis tenham sido cobertos.
5. Obtenção das tabelas de resultados (Tabelas 6.3 a 6.7) e das tabelas de tempo de avaliação (Tabelas 6.8 a 6.12).

Se algum elemento requerido for considerado não executável pelo usuário, a causa de sua não executabilidade será anotada. Caso algum elemento requerido executável não seja executado por questões de não-determinismo, o usuário deverá alterar manualmente a seqüência de sincronizações desejadas para que o elemento requerido em questão seja coberto.

6.3 Programas Utilizados

6.3.1 MDC

Este programa paralelo, do tipo mestre-escravo, calcula o máximo divisor comum entre três números inteiros [30] que são passados na linha de comando. Quatro tarefas paralelas devem ser utilizadas neste experimento, uma tarefa mestre coordena os resultados das três tarefas escravo, cada escravo calcula o m.d.c entre dois números. A comunicação é bloqueante e são utilizados *receives* não-determinísticos.

6.3.2 PI

Este programa paralelo, do tipo mestre-escravo, calcula uma aproximação do número π através do algoritmo *dash board* [17]. Todos os processos calculam uma aproximação de π e um dos processos faz a média. Este programa funciona para qualquer número de tarefas, porém convencionou-se o uso de três tarefas para facilitar as condução do experimento. Todas as tarefas realizam computação e não há dados de entrada. A comunicação é bloqueante e são utilizados *receives* não-determinísticos.

6.3.3 Jantar dos Filósofos

Este programa paralelo simula o compartilhamento de recursos através do problema clássico: o jantar dos filósofos [14]. Cinco filósofos concorrem pelo acesso aos garfos (recurso compartilhado). Como entrada é passado um inteiro, que representa a fome de cada filósofo. Cada filósofo é representado por um processo paralelo e os garfos por um outro processo. A comunicação é bloqueante e são utilizados *receives* não-determinísticos.

6.3.4 Multiplicação de Matrizes

Este programa paralelo multiplica duas matrizes através da decomposição do domínio [42]. O usuário entra com as dimensões e valores em ponto flutuante via teclado na tarefa mestre. Apesar desse problema ser escalável para n processos, convencionou-se o uso de quatro tarefas para facilitar a condução do experimento. A tarefa mestre envia a partes das matrizes a serem multiplicadas aos escravos, que realizam a computação e devolvem o resultado à tarefa mestre. A comunicação é bloqueante e determinística.

6.3.5 Produtor-Consumidor

Este programa paralelo, do tipo produtor-consumidor, faz uma simulação de processos de produção e consumo de material [16]. Múltiplos processos produtores abastecem um processo consumidor. Convencionou-se o uso de três tarefas produtoras para facilitar a condução do experimento. O usuário deve entrar com a quantidade produzida pelos produtores. A comunicação é não-bloqueante e determinística, a operação de teste de recebimento de mensagem pode influenciar no determinismo da ordem de avaliação das mensagens recebidas.

6.4 Resultados dos Experimentos

As Tabelas 6.3 a 6.7 apresentam, para cada critério, o número de elementos requeridos, a cobertura e efetividade dos casos de teste iniciais, a cobertura e efetividade final atingida e a quantidade de elementos não executáveis. A efetividade dos casos de teste refere-se aos casos de teste que contribuiram para aumentar a cobertura do critério avaliado. Os critérios totalmente cobertos pelo conjunto T_{ini}, ou os que tiveram todos seus elementos requeridos executáveis cobertos, não foram reavaliados para o conjunto T e têm as colunas de cobertura e efetividade final em branco, pois estes não se alteram.

Por exemplo, o critério todos-s-usos da Tabela 6.3 foi testado com um conjunto T_{ini} de cardinalidade 99, dos quais sete casos cobriram 24,24% dos 66 elementos requeridos. Os elementos requeridos não cobertos foram analisados e foram necessários três casos de teste adicionais para atingir a cobertura 28,79% com dez casos de teste efetivos. Os demais elementos requeridos mostraram-se não executáveis. Já o critério todos-nós teve cobertura de 100% dos 62 elementos requeridos com o conjunto T_{ini} e portanto foi desnecessário reavaliar o critério com outros dados de teste.

Tabela 6.3: Resultados obtidos para o experimento MDC

Critério	Elementos requeridos	Cobertura inicial $\|T_{ini}\| = 99$	Casos efetivos $\|T_{ini}\| = 99$	Cobertura final $\|T\| = 102$	Casos efetivos $\|T\| = 102$	Elem. não exec.
todos-nós	62	100%	6			0
todos-nós-r	7	100%	2			0
todos-nós-s	7	100%	2			0
todas-arestas	41	51,22%	3			20
todas-arestas-s	30	33,33%	3			20
todos-c-usos	29	100%	6			0
todos-p-usos	40	95%	7	100%	9	0
todos-s-usos	66	24,24%	7	28,79%	10	47

Tabela 6.4: Resultados obtidos para o experimento PI

Critério	Elementos requeridos	Cobertura inicial $\|T_{ini}\| = 1$	Casos efetivos $\|T_{ini}\| = 1$	Cobertura final $\|T\| = 1$	Casos efetivos $\|T\| = 1$	Elem. não exec.
todos-nós	99	63,64%	1			36
todos-nós-r	3	33,33%	1			2
todos-nós-s	3	66,67%	1			1
todas-arestas	27	44,44%	1			15
todas-arestas-s	6	33,33%	1			4
todos-c-usos	69	40,58%	1			41
todos-p-usos	48	47,92%	1			25
todos-s-usos	6	33,33%	1			4

Tabela 6.5: Resultados obtidos para o experimento jantar dos filósofos

Critério	Elementos requeridos	Cobertura inicial $\|T_{ini}\| = 15$	Casos efetivos $\|T_{ini}\| = 15$	Cobertura final $\|T\| = 16$	Casos efetivos $\|T\| = 16$	Elem. não exec.
todos-nós	176	100%	2			0
todos-nós-r	11	100%	1			0
todos-nós-s	36	100%	2			0
todas-arestas	356	21,35%	2			280
todas-arestas-s	325	13,85%	2			280
todos-c-usos	50	100%	2			0
todos-p-usos	148	78,38%	3	81,76%	4	27
todos-s-usos	335	16,42%	2			280

Tabela 6.6: Resultados obtidos para o experimento multiplicação de matrizes

Critério	Elementos requeridos	Cobertura inicial $\|T_{ini}\| = 1$	Casos efetivos $\|T_{ini}\| = 1$	Cobertura final $\|T\| = 3$	Casos efetivos $\|T\| = 3$	Elem. não exec.
todos-nós	368	45,11%	1	45,65%	2	200
todos-nós-r	36	58,33%	1			15
todos-nós-s	36	41,67%	1			21
todas-arestas	1032	4,75%	1	4,84%	2	982
todas-arestas-s	972	2,78%	1			945
todos-c-usos	572	39,34%	1	41,08%	2	337
todos-p-usos	304	30,59%	1	34,21%	2	206
todos-s-usos	1404	1,92%	1	2,07%	3	1375

Tabela 6.7: Resultados obtidos para o experimento produtor-consumidor

Critério	Elementos requeridos	Cobertura inicial $\|T_{ini}\| = 1$	Casos efetivos $\|T_{ini}\| = 1$	Cobertura final $\|T\| = 3$	Casos efetivos $\|T\| = 3$	Elem. não exec.
todos-nós	60	96,67%	1	100%	2	0
todos-nós-r	2	50%	1	100%	2	0
todos-nós-s	3	100%	1			0
todas-arestas	21	80,95%	1	100%	2	0
todas-arestas-s	6	50%	1	100%	2	0
todos-c-usos	43	86,05%	1	95,35%	2	2
todos-p-usos	42	78,57%	1	95,24%	3	2
todos-s-usos	6	50%	1	100%	2	0

6.5 Análise dos Resultados

Para o experimento MDC foram gerados aleatoriamente 99 casos de teste iniciais. A Tabela 6.3 mostra que esses casos de teste cobriram todos os elementos requeridos para alguns dos critérios, porém, dois casos de teste adicionais foram necessários para cobrir todos os elementos requeridos do critério todos-p-usos e mais dois casos casos de teste para atingir a cobertura de 28,79% para o critério todos-s-usos. Foi necessário usar o mecanismo de execução controlada para atingir essa taxa de cobertura no critério todos-

s-usos. As vinte arestas de comunicação não cobertas para o critério todas-arestas-s são as mesmas não cobertas para o critério todas-arestas e referem-se a sincronizações não executáveis.

Para o experimento PI foi gerado manualmente um caso de teste inicial. A Tabela 6.4 mostra que esse caso de teste cobriu todos os elementos requeridos executáveis em todos os critérios. Isto se deve a dois fatores: não há entradas para o programa e o determinismo das sincronizações.

Para o experimento jantar dos filósofos foram gerados aleatoriamente 15 casos de teste iniciais. A Tabela 6.5 mostra que apenas o critério todos-p-usos teve elementos requeridos executáveis não cobertos por T_{ini}. Foi necessário analisar os elementos requeridos não cobertos para a entrada de um caso de teste que cobrisse os elementos requeridos executáveis restantes. Coincidentemente, o número de elementos requeridos não executáveis foi de 280 para os critérios: todas-arestas, todas-arestas-s e todos-s-usos e corresponde às sincronizações não executáveis.

Para o experimento multiplicação de matrizes foi gerado manualmente um caso de teste inicial. A Tabela 6.6 mostra que este caso de teste inicial cobriu os elementos requeridos para os critérios: todos-nós-r, todos-nós-s e todas-arestas-s. Os critérios todos-nós, todas-arestas, todos-c-usos e todos-p-usos tiveram todos os seus elementos requeridos executáveis cobertos por dois casos de teste e o critério todos-s-usos por três casos de teste. O grande número de elementos requeridos para os critérios todas-arestas-s e todas-arestas deve-se ao número de nós com *sends* combinados com os nós de *receive* dos outros processos. Nesse caso, os nós *send* em quatro processos são combinados com nós *receive* dos três outros processos, o que resulta em: $9 \times 4 \times 9 \times 3 = 972$ elementos requeridos. Analogamente para os critérios todas-arestas e todos-s-usos.

Para o experimento produtor-consumidor foi gerado manualmente um caso de teste inicial. A Tabela 6.7 mostra que este caso de teste inicial cobriu todos os nós com *sends*. Os outros critérios, salvo o critério todos-p-usos, obtiveram a cobertura de todos os seus elementos requeridos executáveis com a entrada de um caso de teste adicional. Para atingir a cobertura de 95,24% para o critério todos-p-usos foi necessário alterar o número de vezes que a função MPI_Test é chamada até retornar o indicativo de que a mensagem requisitada foi recebida, isto é possível através da inclusão deste número no arquivo de seqüencia de sincronizações do processo que faz a requisição.

É importante notar que mesmo que a comunicação seja determinística, ou seja, os *receives* identificam os *sends* correspondentes, a seqüência de instruções poderia ter sido

diferente em alguma execução do experimento produtor-consumidor se não fosse o algoritmo que substitui a função MPI_Test original (Apêndice A) e portanto, o caso de teste poderia deixar de cobrir um dos elementos requeridos do critério todos-p-usos que avalia o teste de recebimento da mensagem.

As Tabelas 6.8 a 6.12 comparam os custos de avaliação da cobertura de cada critério para o conjunto T, ou T_{ini} se o critério não tiver sido reavaliado, em relação ao tempo total (do processo do usuário no processador) da avaliação e o tempo por elemento requerido. Note que a relação tempo X elemento coberto não pode ser considerada exata pois, um elemento requerido é considerado coberto ao primeiro caso de teste que o cobrir, ou seja, os casos de teste efetivos, desconsiderando assim os demais casos de teste que também poderiam cobrí-lo. A avaliação da cobertura foi feita na máquina aquarius (dados na Tabela 6.1).

A grande quantidade de elementos requeridos não executáveis nos experimentos PI e multiplicação de matrizes deveu-se, principalmente, ao fato de que os processos paralelos testados referiam-se a uma mesma função. Os diferentes caminhos executados nesta função, em cada processo, são resultado de instruções de desvio de fluxo baseados no número do processo. Desta forma vários nós de um dado processo não podem ser cobertos e conseqüentemente os elementos requeridos que requerem esses nós são não executáveis.

Nos experimentos realizados o tempo de avaliação dos casos de teste para a cobertura de um elemento requerido apresentou pouca variação para os critérios: todos-nós, todos-nós-r e todos-nós-s. A diferença também é pequena entre a avaliação dos critérios: todas-arestas-s e todas-arestas, pois o primeiro é subconjunto do segundo. A diferença no tempo de avaliação entre os critérios todos-c-usos e todos-p-usos só é significativa no experimento jantar dos filósofos.

Em todos os casos o critério mais custoso foi o critério todos-s-usos, que chegou a demorar aproximadamente 25 segundos por elemento requerido na avaliação do experimento jantar dos filósofos. Essa discrepância deveu-se principalmente à combinação entre os processos, que são avaliados dois-a-dois, portanto deve-se esperar uma proporção temporal pelo menos $\binom{n}{2}$ maior, onde n é o número de processos em paralelo.

Para diminuir o tempo de avaliação dos critérios todas-arestas, todas-arestas-s e todos-s-usos o usuário poderia retirar da lista de elementos requeridos aquelas arestas que não puderem ser identificadas como executáveis, principalmente devido à combinação de sincronizações entre os processos.

Tabela 6.8: Custos de avaliação para o experimento MDC

Critério	Tempo total (s)	Tempo por elemento (s)
todos-nós	0,109	0,002
todos-nós-r	0,015	0,002
todos-nós-s	0,013	0,002
todas-arestas	1,907	0,047
todas-arestas-s	1,890	0,063
todos-c-usos	0,328	0,011
todos-p-usos	0,734	0,018
todos-s-usos	52,442	0,794

Tabela 6.9: Custos de avaliação para o experimento PI

Critério	Tempo total (s)	Tempo por elemento (s)
todos-nós	1,754	0,018
todos-nós-r	0,063	0,021
todos-nós-s	0,050	0,017
todas-arestas	0,506	0,019
todas-arestas-s	0,097	0,016
todos-c-usos	1,245	0,018
todos-p-usos	0,848	0,018
todos-s-usos	0,557	0,093

Tabela 6.10: Custos de avaliação para o experimento jantar dos filósofos

Critério	Tempo total (s)	Tempo por elemento (s)
todos-nós	15,211	0,086
todos-nós-r	0,658	0,060
todos-nós-s	1,711	0,047
todas-arestas	298,663	0,839
todas-arestas-s	294,388	0,905
todos-c-usos	4,637	0,093
todos-p-usos	128,679	0,869
todos-s-usos	8432,100	25,170

Tabela 6.11: Custos de avaliação para o experimento multiplicação de matrizes

Critério	Tempo total (s)	Tempo por elemento (s)
todos-nós	5,503	0,015
todos-nós-r	0,260	0,007
todos-nós-s	0,234	0,006
todas-arestas	10,183	0,010
todas-arestas-s	4,693	0,005
todos-c-usos	7,400	0,013
todos-p-usos	5,331	0,017
todos-s-usos	197,974	0,141

Tabela 6.12: Custos de avaliação para o experimento produtor-consumidor

Critério	Tempo total (s)	Tempo por elemento (s)
todos-nós	0,033	0,001
todos-nós-r	0,006	0,003
todos-nós-s	0,007	0,002
todas-arestas	0,019	0,001
todas-arestas-s	0,013	0,002
todos-c-usos	0,039	0,001
todos-p-usos	0,042	0,001
todos-s-usos	0,041	0,007

7 Conclusão

Este capítulo conclui a dissertação apresentando as contribuições trazidas pelo trabalho realizado e possíveis melhorias através de trabalhos futuros.

7.1 Contribuições

Este trabalho teve por objetivo a implementação de uma ferramenta para teste de programas paralelos desenvolvidos na linguagem C, que utilizam o padrão de passagem de mensagem MPI, denominada ValiMPI. Essa ferramenta implementou os critérios estruturais, baseados em fluxo de controle e fluxo de dados, específicos para programas paralelos em ambiente de passagem de mensagem propostos no escopo do projeto ValiPVM [60].

A ValiMPI implementa os seguintes critérios: todos-nós, todos-nós-r, todos-nós-s, todas-arestas, todas-arestas-s, todos-c-usos, todos-p-usos e todos-s-usos. O critério mais custoso foi, nos experimentos realizados, o critério todos-s-usos. Porém, o usuário pode diminuir o tempo de avaliação do critério, se as sincronizações forem determinísticas, removendo da lista de elementos requeridos os elementos que requerem arestas interprocessos não executáveis, quando possível determiná-las estaticamente.

A ValiMPI possibilita ao usuário a execução controlada dos casos de teste, fornece os elementos requeridos pelos critérios e realiza a análise de adequação do conjunto de testes fornecendo a medida de cobertura. A contribuição da ferramenta é evidenciada pela Tabela 7.1, que compara a ValiMPI às outras ferramentas disponíveis para teste de programas paralelos. O principal destaque da ValiMPI em relação às outras ferramentas é o apoio aos critérios baseados em fluxo de dados e fluxo de controle no paradigma de passagem de mensagem.

Os experimentos do capítulo anterior demonstram a viabilidade da aplicação dos critérios propostos pelo projeto ValiPVM em problemas reais e o auxílio proporcionado pela ValiMPI na cobertura do código, o que possibilita quantificar a atividade de teste e

Tabela 7.1: Comparativo entre as ferramentas da Seção 4.6 e a ValiMPI

Ferramenta	Ambiente ou linguagem	Critérios fluxo de dados	Critérios fluxo de contr.	Determinismo da execução	Monitoramento ou visualização	Análise de desempenho
TDC Ada	Ada			√		
ConAn	Java			√		
Della Pasta	Mem.comp.	√		√	√	
Xab	PVM				√	
XPVM	PVM				√	
Visit	PVM				√	√
MDB	PVM			√	√	
STEPS	PVM		√	√	√	
Astral	PVM		√		√	√
Paragraph	MPI				√	√
XMPI	MPI				√	
Umpire	MPI				√	
ValiMPI	MPI	√	√	√	√	

dizer o quão bom é um caso de teste ou se o programa foi testado suficientemente.

7.2 Trabalhos Futuros

Possíveis melhorias à ValiMPI incluem:

- Criar uma IDel.C mais completa em relação à gramática da linguagem C.
- Adaptação da ferramenta às linguagens Fortran e C++.
- Implementação dos critérios: todas-defs, todas-defs/s, todos-s-c-usos e todos-s-p-usos.
- Edição de um manual de usuário e simplificação da sintaxe de linha de comando (Apêndice B).
- Desenvolvimento de uma interface gráfica intuitiva para facilitar a aplicação e visualização dos testes.
- Adaptação da ferramenta a contextos de comunicação diferentes do contexto global (MPI_COMM_WORLD).
- Possibilidade do usuário manter conjuntos de casos de teste separados por quantidade de processos paralelos (fixos e conhecidos estaticamente), *e.g.* o experimento

multiplicação de matrizes poderia ser feito usando 8 processos paralelos e não apenas 4. Isto implicaria na geração de um conjunto de elementos requeridos diferente para cada quantidade de processos paralelos.

- Aperfeiçoar os critérios comunicacionais para diminuir o número de elementos requeridos não executáveis fáceis de identificar, *e.g. receives* determinísticos que usam números ordinais na identificação do processo emissor e *tag*.

- Adaptação dos critérios para um número dinâmico ou desconhecido de processos paralelos.

- Realização de novos experimentos.

Referências

[1] Al-ladan, M. A survey and a taxonomy of approaches for testing parallel and distributed programs. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 01)* (June 2001), pp. 273–279.

[2] Almasi, G. S., e Gottlieb, A. *Highly Parallel Computing*, 2^{nd} ed. The Benjamin/Cummings Publishing Company, Redwood City, 1994.

[3] Beguelin, A. L. Xab: A tool for monitoring PVM programs. In *Workshop on Heterogeneous Processing (WHP 93)* (April 1993), IEEE Computer Society Press, pp. 92–97.

[4] Beizer, B. *Software Testing Techniques*, 2^{nd} ed. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[5] Browne, S., Dongarra, J., e London, K. Review of performance analysis tools for MPI parallel programs. *NHSE Review* (1998).

[6] Bueno, P. M. S., Chaim, M. L., Jino, M., Maldonado, J. C., e Vilela, P. R. *POKE-TOOL – Versão 2.0 – Manual do Usuário*. DCA/FEE/UNICAMP, 1994.

[7] Burns, G., Daoud, R., e Vaigl, J. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium* (1994), pp. 379–386.

[8] Carver, R. H., e Tai, K. C. Replay and testing for concurrent programs. *IEEE Software 8*, 2 (1991), 66–74.

[9] Chaim, M. L. Poke tool — uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados. Dissertação (Mestrado), DCA/-FEE/UNICAMP, 1991.

[10] Chung, C. M., Shih, T. K., Wang, Y. H., Lin, W. C., e Kou, Y. F. Task decomposition testing and metrics for concurrent programs. In 5^{th} *International Software Reliability Engineering* (White Plains, NY, USA, 1996), IEEE Computer Society Press, pp. 122–130.

[11] Chusho, T. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Transactions on Software Engineering 13*, 5 (May 1987), 509–517.

[12] Damodaran-Kamal, S. K., e Francioni, J. M. Nondeterminacy: Testing and debugging in message passing parallel programs. In *Proceedings of the* 3^{rd} *ACM/ONR Workshop on Parallel and Distributed Debugging* (San Diego, May 1993), ACM Press, pp. 118–128.

[13] DeMillo, R. A., Lipton, R. J., e Sayward, F. G. Hints on test data selection: Help for the practising programmer. *IEEE Computer Magazine 11*, 4 (April 1978), 34–41.

[14] Dijkstra, E. W. Hierarchical ordering of sequential processes. *Acta Inf. 1* (1971), 115–138.

[15] Flynn, M. Some computer organization and their effectiveness. *IEEE Transactions on Computers C-21* (1972), 948–960.

[16] Foster, I. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[17] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., e Walker, D. W. *Solving Problems on Concurrent Processors: General Techniques and Regular Problems*, vol. 1. Prentice–Hall, 1988.

[18] Frankl, P., e Weyuker, E. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering 14*, 10 (October 1988), 1483–1498.

[19] Geist, A., Beguelin, A. L., Dongarra, J., Jiang, W., Manchek, R., e Sunderam, V. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, London, England, 1994.

[20] Goodenough, J. B., e Gerhart, S. L. Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software* (1975), pp. 493–510.

[21] Grady, R. B. Practical results from measuring software quality. *Communications of the ACM 36*, 11 (1993), 62–68.

[22] Gropp, W. D., e Lusk, E. Goals guiding design: PVM and MPI. In *Proceedings of IEEE Cluster* (2002), IEEE Computer Society Press, pp. 257–265.

[23] Hansen, P. B. Testing a multiprogramming system. *Software – Practice and Experience 3* (1973), 145–150.

[24] Heath, M. T., e Etheridge, J. A. Visualizing the performance of parallel programs. *IEEE Softw. 8*, 5 (1991), 29–39.

[25] Hwang, K., e Xu, Z. *Scalable Parallel Computing*, 2^{nd} ed. McGraw-Hill, Boston, US, 1998.

[26] IEEE. IEEE Standard Glossary of Software Engineering Terminology, 1990. IEEE Std. 610.12-1990.

[27] Ilmberger, H., Thürmel, S., e Wiedemann, C. P. Visit: A visualization and control environmenent for parallel program debugging. In *Proceedings of the 3^{rd} ACM/ONR Workshop on Parallel and Distributed Debugging* (San Diego, California, May 1993), pp. 199–201.

[28] Kohl, J. A., e Geist, G. A. *XPVM 1.0 User's Guide.* Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA, November 1996.

[29] Koppol, P. V., Carver, R. H., e Tai, K. C. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering 28*, 6 (June 2002), 607–623.

[30] Krawczyk, H., e Wiszniewski, B. Classification of software defects in parallel programs, 1994. Tech. Report 2.

[31] Leveson, N., e Turner, C. S. An investigation of the Therac-25 accidents. *IEEE Computer 26*, 7 (July 1993), 18–41.

[32] Long, B., Hoffman, D., e Strooper, P. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering 29*, 6 (June 2003), 555–565.

[33] Lourenco, J., Cunha, J., Krawczyk, H., Kuzora, P., Neyman, M., e Wiszniewski, B. An integrated testing and debugging environment for parallel and distributed programs. In *Proceedings of the 23rd EUROMICRO Conference* (Budapest, Hungary, September 1997), IEEE Computer Society Press, pp. 291–298.

[34] Maldonado, J. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese (Doutorado), DCA/FEE/UNICAMP, 1991.

[35] McCabe, T. J., e Butler, C. W. Design complexity measurement and testing. *Communications of the ACM 32*, 12 (December 1989), 1415–1425.

[36] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[37] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[38] Moreira, E. M., Santana, R. H. C., Santana, M. J., e Sant'Ana, T. D. Auxiliary tool for debugging parallel programs. In *Proceedings of the 3rd International Information and Telecommunication Technologies Symposium* (São Carlos, SP, 2004).

[39] Myers, G. J. *The Art of Software Testing*. John Wiley, 1979.

[40] Neyman, M., Krawczyk, H., Kuzora, P., Proficz, J., e Wiszniewski, B. STEPS - a tool for testing PVM programs. In *Proceedings of the 3rd SEIHPC Workshop* (Madrid, Spain, January 1998).

[41] Pressman, R. S. *Software Engineering: A Practitioner's Approach*, 5^{th} ed. McGraw-Hill, 2001.

[42] Quinn, M. J. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, Boston, US, 1987.

[43] Rapps, S., e Weyuker, E. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering 11*, 04 (April 1985), 367–375.

[44] Simão, A. S., Vicenzi, A. M. R., Maldonado, J. C., e Santana, A. C. L. Software product instrumentation description. Relatório técnico, Universidade de São Paulo – Campus São Carlos, 2002.

[45] Simão, A. S., Vicenzi, A. M. R., Maldonado, J. C., e Santana, A. C. L. A language for the description of program instrumentation and automatic generation of instrumenters. *CLEI Electronic Journal 6*, 1 (December 2003).

[46] Souza, P. S. L. Máquina paralela virtual em ambiente windows. Dissertação (Mestrado), ICMC/USP, maio 1996.

[47] Souza, S. R. S., Vergilio, S. R., Souza, P. S. L., Simão, A. S., Gonçalves, T. B., Lima, A. M., e Hausen, A. C. Valipar: A testing tool for message-passing parallel programs. In *17th International Conference on Software Engineering and Knowledge Engineering (SEKE 2005)* (Taipei, Taiwan, July 2005), pp. 386–390.

[48] Squyres, J. M., e Lumsdaine, A. A component architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting* (Venice, Italy, September 2003), no. 2840 in Lecture Notes in Computer Science, Springer-Verlag, pp. 379–387.

[49] Tai, K. C. On testing concurrent programs. In *Proceedings of COMPSAC* (October 1985), pp. 310–317.

[50] Tai, K. C., Carver, R. H., e Obaid, E. E. Debugging concurrent ada programs by deterministic execution. *IEEE Trans. Softw. Eng. 17*, 1 (1991), 45–63.

[51] Tanenbaum, A. S. *Distributed Operating Systems*. Prentice Hall, 1995.

[52] Tanenbaum, A. S. *Modern Operating Systems*, 2nd ed. Prentice Hall, 2001.

[53] Tassey, G. The economic impacts of inadequate infrastructure for software testing. Relatório técnico, National Institute of Standards and Technology, May 2002. RTI Project Number 7007.011.

[54] Taylor, R. N. General-purpose algorithm for analyzing concurrent programs. *Communications of the ACM 26* (May 1983), 361–376.

[55] Taylor, R. N., Levine, D. L., e Kelly, C. D. Structural testing of concurrent programs. *IEEE Transactions on Software Enginnering 18*, 3 (March 1992).

[56] The LAM/MPI Team. XMPI. Open Systems Laboratory, Indiana University. Disponível em: `www.lam-mpi.org/beta/`.

[57] The LAM/MPI Team. *LAM/MPI User's guide*. Open Systems Laboratory, Indiana University, September 2004.

[58] Torácio, A. A. Definição e implementação de um módulo de suporte à aplicação de critérios de teste em ambientes de passagem de mensagem. Relatório técnico, DInf - UFPR, Curitiba, PR, Dezembro 2004. Relatório de Iniciação Científica. Projeto ValiPar - PDPGTI-CNPq.

[59] unix. *unix*, vol. unix. unix, unix.

[60] Vergilio, S. R., Souza, S. R. S., e Souza, P. S. L. Coverage testing criteria for message-passing parallel programs. In *6th IEEE Latin American Test Workshop (LATW)* (Salvador, Bahia, March 2005), pp. 161–166.

[61] Vetter, J. S., e Supinski, B. R. Dynamic software testing of MPI applications with Umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (2000), IEEE Computer Society Press, p. 51.

[62] Weiss, S. N. A formal framework for the study of concurrent program testing. In *Proceedings of the 2nd Workshop on Software Testing, Analysis and Verification* (July 1988), pp. 106–113.

[63] Yang, C. S. *Progam-Based Structured Testing of Shared Memory Parallel Programs.* Tese (Doutorado), University of Delaware, 1999.

[64] Yang, C.-S., e Pollock, L. The challenges in automated testing of multithreaded programs. In *14th International Conference on Testing Computer Software* (June 1997).

[65] Yang, C. S. D., Souter, A. L., e Pollock, L. L. All-du-path coverage for parallel programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (1998).

[66] Yang, R. D., e Chung, C. G. Path analysis testing of concurrent programs. *ACM Information and Software Technology 34*, 1 (January 1992).

APÊNDICE A – Algoritmos

Este apêndice descreve os principais algoritmos, na forma de pseudo-código, utilizados pela biblioteca ValiMPI.

A.1 *Check-points* do Traço de Execução

A função de *check-point*, que escreve pares (nó,processo) no arquivo de traço de execução, é na verdade uma o controle de uma fila desses pares, que são escritos em arquivo.

Algoritmo 1 ValiMPI_Check_trace(**input:** fila_traço, nó)

```
{Check-point}
mpi_inicializado ⇐ MPI_Initialized( )
if mpi_inicializado and escreve_no_arquivo_traço then
  while not Is_Empty(fila_traço) do
    temp ⇐ Pop(fila_traço)
    p ⇐ temp.processo
    if p = –1 then
      p ⇐ processo_atual
    end if
    File_Write(arquivo_traço, temp.nó, p)
  end while
  File_Write(arquivo_traço, nó, processo_atual)
else
  Push(fila_traço, nó, –1)
end if
```

A.2 Envio de Mensagem

As operações de envio de mensagem bloqueante e não-bloqueante são análogas, o nó e a mensagem original são empacotados em uma nova mensagem que é enviada (de maneira bloqueante ou não-bloqueante).

Algoritmo 2 ValiMPI_Send(**input:** mensagem, destino, tag)

```
{Send bloqueante, análogo para não-bloqueante}
Empacota(nó_send, nova_mensagem)
Empacota(mensagem, nova_mensagem)
MPI_Send(nova_mensagem)
```

A.3 Recebimento de Mensagem

A.3.1 Modo Bloqueante

A operação de recebimento de mensagem no modo bloqueante tem comportamento distinto durante a execução controlada e a não-controlada.

- Na execução não-controlada uma mensagem (recebida de maneira bloqueante) tem as informações de qual nó originou o envio e a mensagem originalmente enviada desempacotadas. Os pares (nó,processo) do envio e (nó,processo) do recebimento são enfileirados na fila de traço de execução. A sincronização realizada é enfileirada na fila de seqüências de sincronização.

- A execução controlada supõe a existência de uma fila com todas as sincronizações previstas, que é usada na operação de recebimento de mensagem. O procedimento é semelhante ao caso da execução não-controlada, porém não enfileira na fila de sincronizações.

Algoritmo 3 ValiMPI_Recv(**output:** mensagem; **input:** origem, tag)

```
{Receive bloqueante}
if execução_controlada = true then
   temp ⇐ Pop(fila_sincronizações)
   origem ⇐ temp.origem
   tag ⇐ temp.origem
end if
MPI_Recv_bloqueante(mensagem_recebida, origem, tag)
Desempacota(mensagem_recebida, nó_send)
Desempacota(mensagem_recebida, mensagem)
Push(fila_traço, nó_send, origem)
Push(fila_traço, nó_recv, destino)
if not execução_controlada then
   Push(fila_sincronizações, nó_recv, nó_send, origem, tag)
end if
```

A.3.2 Modo Não-Bloqueante

A operação de recebimento de mensagem de modo não-bloqueante está fortemente acoplada à operação de teste de recebimento de mensagem.

A.3.2.1 Requisição de Recebimento de Mensagem

A operação de recebimento não-bloqueante de mensagem desabilita a escrita no arquivo de traço de execução e, no caso da execução não-controlada, um nó de sincronização "*dummy*" é enfileirado. Um nó de traço de execução "*dummy*" também é enfileirado. São guardadas, numa lista de requisições, as referências para esses nós "*dummies*" para posterior preenchimento. O nó da operação de recebimento não bloqueante é enfileirado na fila de traço de execução e, no caso da execução controlada, as informações para a próxima sincronização é desenfileirada para uso na operação de recebimento de mensagem.

Algoritmo 4 ValiMPI_Irecv(**output:** mensagem, requisição; **input:** origem, tag)

```
{Receive não-bloqueante}
escreve_no_arquivo_traço ⇐ false
if not execução_controlada then
   {reserva espaço para a sincronização}
   nó_sincronização ⇐ Push(fila_sincronizações, 0, 0, 0, 0)
end if
{reserva espaço para o traço de execução}
nó_traço ⇐ Push(fila_traço, 0, 0)
Insert(lista_requisições, requisição, nó_sincronização, nó_traço, nó_recv)
Push(fila_traço, nó_recv, destino)
if execução_controlada = true then
   temp ⇐ Pop(fila_sincronizações)
   origem ⇐ temp.origem
   tag ⇐ temp.tag
end if
MPI_Recv_não_bloqueante(mensagem, requisição, origem, tag)
```

A.3.2.2 Teste de Recebimento de Mensagem

A operação de teste procura pela requisição passada como parâmetro de entrada em uma lista e não for uma execução controlada o contador de chamadas da função de teste (inicializado em zero) é incrementado e o teste de recebimento é feito, se for uma execução controlada o contador de chamadas da função de teste (inicializado com o número de chamadas da execução anterior) é decrementado e se ele chegar a zero espera-se o recebimento da mensagem.

Este algoritmo age desse modo para garantir o determinismo da seqüência de instruções executas. Isto também pode possibilitar uma cobertura maior dos critérios todos-p-usos e todas-arestas sem a necessidade de executar o programa um número indefinido de vezes, basta editar o número de vezes que a função de teste deve ser chamada no arquivo de seqüência de sincronizações.

Se a mengem tiver sido recebida, no modo de execução controlada, o nó "*dummy*" da fila de sincronizações, relativo à requisição atendida, é preenchido. Em ambos modos, o nó "*dummy*" da fila de traço de execução também é preenchido. A requisição atendida é removida da fila e, se a lista ficar vazia, a escrita no arquivo de traço de execução é reabilitada. Finalmente, retorna-se o valor booleano de recebimento da mensagem.

Algoritmo 5 ValiMPI_Test(**input:** requisição): **boolean**

```
{Teste de recebimento de mensagem}
recebeu ⇐ false
nó_requisição ⇐ Search(lista_requisições, requisição)
if not execução_controlada then
  Incrementa(nó_requisição.num_chamadas_mpi_test)
  recebeu ⇐ MPI_Test(requisição)
else
  Decrementa(nó_requisição.num_chamadas_mpi_test)
  if nó_requisição.num_chamadas_mpi_test = 0 then
    MPI_Wait(requisição)
    recebeu ⇐ true
  end if
end if
if recebeu then
  Desempacota(mensagem_recebida, nó_send)
  Desempacota(mensagem_recebida, mensagem)
  if not execução_controlada then
    {preenche sincronização reservada}
    Set_Node(nó_requisição.nó_sincronização, nó_recv, nó_send, origem, tag)
  end if
  {preenche traço reservado}
  Set_Node(nó_requisição.nó_traço, nó_send, origem)
  Remove(lista_requisições, requisição)
  if Is_Empty(lista_requisições) then
    escreve_no_arquivo_traço ⇐ true
  end if
end if
return recebeu
```

APÊNDICE B – Sintaxe dos Módulos

Este apêndice apresenta a sintaxe de cada módulo acompanhada de um exemplo de uso.

B.1 Instrumentador e Grafos definição-uso

Sintaxe: vali_inst ⟨arquivo.c⟩

Exemplo: vali_inst mdc.c

Saída: arquivos mdc.c_instrumentado.c, C.mestre.dot, C.escravo.dot e C.main.dot.

B.1.1 Compilador

Sintaxe: vali_cc ⟨parâmetros do compilador⟩

Exemplo: vali_cc -Wall mdc.c_instrumentado.c -o mcd_instrumentado

Saída: arquivo executável mcd_instrumentado.

B.2 Gerador de Elementos Requeridos

Sintaxe: vali_elem ⟨n⟩ ⟨"f(x)" ["g(y)"...]⟩

Onde: n é o número de processos.

f é o nome de uma função.

x é uma lista de processos.

Exemplo: vali_elem 4 "mestre(0)" "escravo(1,2,3)"

Saída: elementos requeridos e descritores dos critérios no diretório ./valimpi/res/

B.3 Executor

B.3.1 Execução Não-Controlada

Sintaxe: vali_exec ⟨número do caso de teste⟩ run ⟨número de processos⟩ executável " ⟨argumentos⟩"

Exemplo: vali_exec 1 run 4 mcd_instrumentado "12 44 36"

Saída: saída do programa mcd_instrumentado na tela. Os arquivo seguintes arquivos são criados no diretório ./valimpi/test_case0001: nome do executável em progname, saída padrão em stdout, saída de erro stderr, argumentos de entrada em args, entrada do teclado em stdin, arquivos de traço de execução (trace.[main|mestre|escravo].p[0...3]), arquivos de seqüência de sincronização seq_sync.p[0...3] e xmpi.lamtr (se a implementação LAM/MPI estiver disponível).

B.3.2 Execução Controlada

Sintaxe: vali_exec ⟨número do caso de teste⟩ rerun

Exemplo: vali_exec 1 rerun

Saída: as saídas da execução não-controla servem de entrada na execução controlada, exceto os arquivos de traço de execução.

B.4 Avaliador

Sintaxe: vali_eval ⟨critério⟩ ⟨n⟩ ⟨"f(x)" ["g(y)"...]⟩ Critérios disponíveis: todos-nos, todos-nosR, todos-nosS, todas-arestas, todas-arestasS, todos-c-usos, todos-p-usos e todos-s-usos.

Exemplo: vali_eval todos-nosR 4 "mestre(0)" "escravo(1,2,3)"

Saída: lista dos elementos requeridos cobertos, não cobertos e percentual de cobertura.