UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ANÁLISE AUTOMÁTICA DE ACESSOS CONCORRENTES A DADOS PARA REFATORAÇÃO DE CÓDIGO SEQUENCIAL EM CÓDIGO PARALELO OPENMP

DISSERTAÇÃO DE MESTRADO

Dionatan Kitzmann Tietzmann

Santa Maria, RS, Brasil

2011

ANÁLISE AUTOMÁTICA DE ACESSOS CONCORRENTES A DADOS PARA REFATORAÇÃO DE CÓDIGO SEQUENCIAL EM CÓDIGO PARALELO OPENMP

por

Dionatan Kitzmann Tietzmann

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Mestre em Computação

Orientador: Profª Drª Andrea Schwertner Charão (UFSM)

Santa Maria, RS, Brasil

2011

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

ANÁLISE AUTOMÁTICA DE ACESSOS CONCORRENTES A DADOS PARA REFATORAÇÃO DE CÓDIGO SEQUENCIAL EM CÓDIGO PARALELO OPENMP

elaborada por
Dionatan Kitzmann Tietzmann

como requisito parcial para obtenção do grau de
Mestre em Computação

COMISSÃO EXAMINADORA:

(Presidente/Co-orientador)

Prof. Dr. Eduardo Kessler Piveta (UFSM)

Prof. Dr. André Rauber du Bois (UFPel)

Santa Maria, 16 de Dezembro de 2011.

AGRADECIMENTOS

Agradeço à minha companheira Laís por sempre me apoiar e pela compreensão nos momentos em que não pude estar presente.

À minha família pelo incentivo.

À professora Andrea Charão que sempre esteve disposta a ajudar, por acreditar no meu potencial e pelas suas orientações e sugestões valiosas ao trabalho.

Aos professores Eduardo Piveta, Jairo Panetta e Benhur Stein pelas contribuições ao trabalho.

Ao meu primo Mário Avelino Kitzmann e à amiga Lilian Delavy por me acolherem em suas residências quando necessitei.

Ao amigo Bruno Boniati pela enorme ajuda e incentivo.

Aos colegas de trabalho da UNIJUÍ por flexibilizarem minhas ausências e pelo apoio dado.

Aos colegas de mestrado, em especial aos amigos Gustavo Rissetti e Vinícius Maran, pela amizade e troca de conhecimento.

“A mente que se abre a uma nova ideia jamais voltará ao seu tamanho original”
— ALBERT EINSTEIN

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

ANÁLISE AUTOMÁTICA DE ACESSOS CONCORRENTES A DADOS PARA REFATORAÇÃO DE CÓDIGO SEQUENCIAL EM CÓDIGO PARALELO OPENMP

Autor: Dionatan Kitzmann Tietzmann
Orientador: Profª Drª Andrea Schwertner Charão (UFSM)
Local e data da defesa: Santa Maria, 16 de Dezembro de 2011.

A transformação manual de programas sequenciais em código paralelo não é uma tarefa fácil. Ela requer muito esforço e atenção do programador durante esse processo, correndo grande risco de se introduzir erros que podem não ser percebidos pelo programador. Um desses problemas, fortemente ligado à programação paralela em memória compartilhada, é a condição de corrida. Esse problema ocorre em virtude da manipulação concomitante realizada por mais de uma *thread* sobre uma variável compartilhada entre elas, sendo o resultado desta variável dependente da ordem de acesso. Explorando essa dificuldade, este trabalho propõe uma abordagem que auxilie o programador durante a refatoração de código sequencial para código paralelo OpenMP, identificando de forma automatizada variáveis que podem vir a ter problemas de condição de corrida. Para tanto, é proposto um algoritmo de verificação baseado no acesso às variáveis e feita a sua implementação utilizando-se do *framework* da ferramenta Photran (um *plugin* para edição de código Fortran integrado ao IDE Eclipse). Para fins de avaliação empírica do algoritmo, apresentam-se testes realizados com pequenos programas e exemplos de código, mostrando o funcionamento da ferramenta nos casos previstos. Além disso, apresenta-se um estudo de caso baseado em uma aplicação real e complexa, mostrando a habilidade do algoritmo em identificar as variáveis em risco, bem como ilustrando algumas de suas limitações conhecidas.

Palavras-chave: Refatoração, programação paralela, condição de corrida, OpenMP.

ABSTRACT

Master's Dissertation
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

AUTOMATIC ANALYSIS OF CONCURRENT ACCESS DATA FOR SEQUENTIAL CODE REFACTORING IN OPENMP PARALLEL CODE

Author: Dionatan Kitzmann Tietzmann
Advisor: Prof[a] Dr[a] Andrea Schwertner Charão (UFSM)

The manual transformation of sequential programs into parallel code is not an easy task. It requires very effort and attention of the developer during this process at great risk of introducing errors that can not be perceived by the programmer. One of these problems, strongly connected to shared memory parallel programming is the race condition. This problem occurs because of the simultaneous manipulation performed for more than a *thread* on a variable shared between them, with the result of this variable dependent of the access order. Exploring this difficulty, this work proposes an approach that helps the programmer during the refactoring of a sequential code for OpenMP parallel code, identifying variables in an automated manner that may have problems of race condition. To this end, we propose a verification algorithm based on access to the variables and made its implementation using the Photran *framework* tool (a *plugin* for editing FORTRAN code integrated into the Eclipse IDE). For purposes of empirical evaluation of the algorithm, we present tests with small programs and code examples showing the operation of the tool in the cases provided. In addition, it presents a case study based on a real and complex application, showing the ability of the algorithm to identify all the variables at risk, as well as illustrating some of its known limitations.

Keywords: refactoring, parallel programing, race condidition, OpenMP.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IDE	*Integrated Development Environment*
API	*Application Programming Interface*
OpenMP	*Open Multi-Processing*
CLR	*Common Language Runtime*
MPI	*Message Passing Interface*
CAPO	*Computer-Aided Parallelizer and Optimizer*
NASA	*National Aeronautics and Space Administration*
CDT	*C/C++ Development Tools*
AST	*Abstract Syntax Tree*
VPG	*Virtual Program Graph*
BLAS	*Basic Linear Algebra Subprograms*
BRAMS	*Brazilian Regional Atmospheric Modeling System*
IMSL	*International Mathematics and Statistics Library*
LAPACK	*Linear Algebra PACKage*
NAG	*Numerical Algorithms Group*
OmpSCR	*OpenMP Source Code Repository*
OLAM	*Ocean-Land Atmosphere Model*
RAMS	*Regional Atmospheric Modeling System*

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto e motivação

A busca por maior capacidade de processamento motivou a evolução das arquiteturas computacionais, resultando em vários tipos de arquiteturas que exploram a noção de paralelismo. Dentre elas, estão as arquiteturas com múltiplos núcleos de processamento ou com múltiplos processadores, que permitem executar tarefas em paralelo e, assim, obter resultados mais rapidamente do que na execução sequencial. Essas arquiteturas possuem em comum outra característica: a memória principal compartilhada entre os núcleos ou entre os processadores.

Em arquiteturas paralelas com memória compartilhada, é comum o uso de um modelo de programação baseado em *threads* (WILKINSON; ALLEN, 2004). Nesse modelo, o programador identifica partes de código de um programa que podem ser executadas em paralelo, ou seja, identifica tarefas a serem executadas por múltiplas *threads*. Cada *thread* representa um fluxo de execução independente associado a um mesmo processo, com dados privados mas também com acesso a dados alocados para o processo, que são compartilhados entre as múltiplas *threads*. Nesse modelo, a comunicação entre as tarefas ocorre através da memória compartilhada, pois uma atualização num dado compartilhado é visível a todas as *threads* do processo.

Para a programação com *threads*, são usadas APIs (*Application Programming Interfaces*) que permitam expressar a decomposição do programa em múltiplas *threads* cooperantes. Nesse contexto, destaca-se a OpenMP (OPENMP, 2011), uma API padrão para programação paralela em arquiteturas com memória compartilhada, mantida por um consórcio de fabricantes de hardware e software. OpenMP facilita a paralelização de um programa por meio de diretivas de compilação. Assim, não é necessário gerenciar *threads* explicitamente, basta indicar um ou mais trechos de código que devem ser executados em

paralelo, com o trabalho dividido entre as *threads*.

Mesmo com as facilidades de OpenMP, a paralelização de um programa existente nem sempre é uma tarefa trivial. O sucesso dessa operação depende de uma análise minuciosa de dados e operações, a fim de identificar o máximo de paralelismo a explorar. Além disso, deve-se analisar os acessos a dados que serão compartilhados entre múltiplas *threads*, a fim de evitar problemas relacionados a acessos concorrentes. Em particular, é importante que o programa paralelizado seja livre de **condições de corrida**. Basicamente, uma condição de corrida é uma situação na qual várias tarefas acessam e manipulam dados compartilhados concorrentemente e sem sincronização, sendo o resultado da execução dependente da ordem em que ocorrem os acessos aos dados (SILBERSCHATZ; GALVIN; GAGNE, 2008; NETZER; MILLER, 1992). Os trechos de código nos quais o acesso concorrente precisa ser controlado são denominados **seções críticas** (TOSCANI; OLIVEIRA; SILVA CARISSIMI, 2003; SILBERSCHATZ; GALVIN; GAGNE, 2008). Sem a devida sincronização, o resultado produzido pode ser incorreto, pois as tarefas cooperantes podem acessar variáveis compartilhadas em qualquer ordem e a qualquer momento, não sendo possível garantir sobre suas velocidades relativas (TOSCANI; OLIVEIRA; SILVA CARISSIMI, 2003).

O padrão OpenMP possui recursos para lidar com a sincronização de tarefas e para controlar o acesso a variáveis compartilhadas. No entanto, cabe ao programador analisar o fluxo de dados e identificar seções críticas decidindo sobre o melhor recurso a utilizar para tratá-las. Nesse cenário, o programador pode se beneficiar de ferramentas que realizam análise automática de acessos a dados no código em questão. Há diversos trabalhos que propõem algoritmos e ferramentas para análise de acessos concorrentes a dados e detecção de condições de corrida (NETZER, 1991; SAVAGE et al., 1997; SEREBRYANY; ISKHODZHANOV, 2009; YU; RODEHEFFER; CHEN, 2005; MICROSYSTEMS, 2007; NAIK; AIKEN; WHALEY, 2006; ENGLER; ASHCRAFT, 2003; BASUPALLI et al., 2011). Em geral, as pesquisas concentram-se na análise dinâmica e/ou na análise estática do programa. A análise dinâmica é feita em tempo de execução, geralmente acompanhando as linhas de execução do programa e registrando seus eventos. Já a análise estática é feita no código-fonte do programa, não durante sua execução.

Grande parte das pesquisas relacionadas tanto à análise dinâmica quanto à análise estática estão voltadas para identificação de problemas em programas já paralelizados.

Analisadores dinâmicos geralmente desfrutam de maior precisão e escalabilidade, entretanto são difíceis de usar em programas em desenvolvimento (NAIK; AIKEN; WHALEY, 2006). Como a análise dinâmica é feita através da execução do programa, são necessários códigos e dados suficientes para realização de testes. Já na análise estática, isso não é necessário pelo fato de a análise se dar sobre o código-fonte do programa, podendo ser feita em vários estágios do projeto, inclusive no início do desenvolvimento.

A transformação de um programa sequencial em um programa paralelo pode ser vista como um conjunto de refatorações (DIG; MARRERO; ERNST, 2009; DIG et al., 2009; WLOKA; SRIDHARAN; TIP, 2009). Entende-se por refatoração uma alteração interna feita num programa, preservando seu comportamento observável (OPDYKE, 1992; FOWLER et al., 1999; TOURWÉ; MENS, 2004). No caso da paralelização de um programa, a refatoração deve garantir que o programa paralelo reproduza os mesmos resultados corretos do programa sequencial original. Nessa linha de raciocínio, a introdução de uma diretiva OpenMP em um código sequencial pode ser vista como uma refatoração.

A refatoração de software vem, há tempos, se beneficiando de ferramentas que automatizam parcial ou totalmente certas transformações de código, previamente catalogadas. De fato, existem muitos ambientes de desenvolvimento integrado (do inglês, *Integrated Development Environment* - IDE) que reúnem refatorações para código sequencial, principalmente para linguagens orientadas a objetos. Tais ambientes provêem infraestruturas para representação de programas em memória, que podem ser usadas como base para análise estática de código. No entanto, há poucos IDEs voltados para a paralelização de código, sendo esse um tema abordado por projetos recentes, ainda em desenvolvimento (PTP, 2011). Além disso, em geral, os trabalhos que visam analisar acessos concorrentes a dados e detectar condições de corrida não se integram a IDEs amplamente utilizados, como por exemplo o IDE Eclipse.

O tema deste trabalho insere-se no contexto recém delineado, abordando a análise de acessos concorrentes a dados como uma etapa para a refatoração automatizada de código sequencial em código paralelo, em arquiteturas paralelas com memória compartilhada.

1.2 Objetivos e metodologia

O principal objetivo deste trabalho é a análise automática de acessos a dados, em trechos de programas sequenciais que deseja-se paralelizar com OpenMP. Essa análise

busca auxiliar o programador a identificar possíveis seções críticas e, assim, ajudá-lo na refatoração de código sequencial em código paralelo. Tal identificação é feita através de um algoritmo proposto que verifica se, em um determinado trecho de código indicado pelo programador, existem variáveis com possibilidade de envolvimento em condições de corrida.

O algoritmo se aplica a programas escritos em linguagens procedurais que, por questões de desempenho, são usadas em muitos programas legados que podem se beneficiar do paralelismo. Esse é o caso da linguagem Fortran, usada em aplicações de computação científica. Essa linguagem foi escolhida como alvo para validação empírica do algoritmo. A implementação do algoritmo é feita em Java, explorando uma infraestrutura para refatoração de código integrada ao IDE Eclipse, denominada Photran (DRAGAN-CHIRILA, 2004; EIPE, 2004; OVERBEY et al., 2005). Cabe destacar que, embora o algoritmo seja implementado usando essa infraestrutura, seu resultado não é uma refatoração de código propriamente dita, mas sim um subsídio para auxiliar o programador na refatoração.

A fim de mostrar como esse subsídio pode ser usado, são sugeridas algumas instruções (diretivas) OpenMP baseadas no resultado da análise. As sugestões ilustram, portanto, refatorações que o programador pode (ou não) realizar para transformar o código sequencial em paralelo. Além disso, as sugestões indicam como este trabalho pode servir de base para outros trabalhos que visem identificar oportunidades de refatoração.

1.3 Organização do texto

Este trabalho está organizado da seguinte forma: o capítulo 2 apresenta uma fundamentação sobre condição de corrida, refatoração e sobre a API OpenMP. Nesse capítulo são abordados conceitos sobre o problema da condição de corrida, métodos utilizados para identificação desse problema e trabalhos que implementam esses métodos. Também são abordados, de forma sucinta, conceitos relacionados à refatoração e à API OpenMP.

O capítulo 3 apresenta o algoritmo de detecção proposto e seus requisitos, aspectos sobre a ferramenta Photran, detalhes sobre a implementação do algoritmo nessa ferramenta e as funcionalidades disponibilizadas pelo detector.

No capítulo 4, é feita a avaliação do detector através de estudos de caso. Por fim, o capítulo 5, apresenta as considerações finais do trabalho, e discute algumas ideias para sua continuidade.

2 FUNDAMENTAÇÃO

Neste capítulo, são apresentados conceitos e estudos relacionados a este trabalho. Primeiramente são apresentadas definições sobre o problema de condição de corrida, em que situação ele ocorre e como pode ser detectado. Na sequencia, são abordadas questões relativas a transformações de código serial em código paralelo, no que tange a paralelização de código usando OpenMP, paralelização automática e semi-automática e o uso de refatoração na paralelização de código. No decorrer do capítulo, são apresentados estudos que estão, de alguma forma, relacionados à abordagem deste trabalho e que têm por objetivo a detecção de problemas de condição de corrida ou a transformação de código serial em código paralelo.

2.1 Concorrência no acesso a dados

2.1.1 Condição de corrida

Condição de corrida, do inglês *race condition*, é a situação na qual vários processos acessam e manipulam dados compartilhados concorrentemente e sem sincronização, sendo o resultado da execução dependente da ordem em que ocorre o acesso aos dados (SILBERSCHATZ; GALVIN; GAGNE, 2008; NETZER; MILLER, 1992). Nesse caso, o resultado da computação, dada uma mesma entrada de dados, pode ter resultados distintos em ocasiões diferentes (TOSCANI; OLIVEIRA; SILVA CARISSIMI, 2003).

A figura 2.1 apresenta um exemplo onde há uma condição de corrida. Nesse exemplo existem duas tarefas, `Tarefa 1` e `Tarefa 2`, compartilhando a variável `count`, cujo valor é `5`. Tais tarefas acessam a variável `count` concomitantemente para executarem suas operações, onde a `Tarefa 1` incrementará a variável `count` e a `Tarefa 2` decrementará esta mesma variável.

Após a execuções das duas operações apresentadas na figura 2.1, o resultado esperado

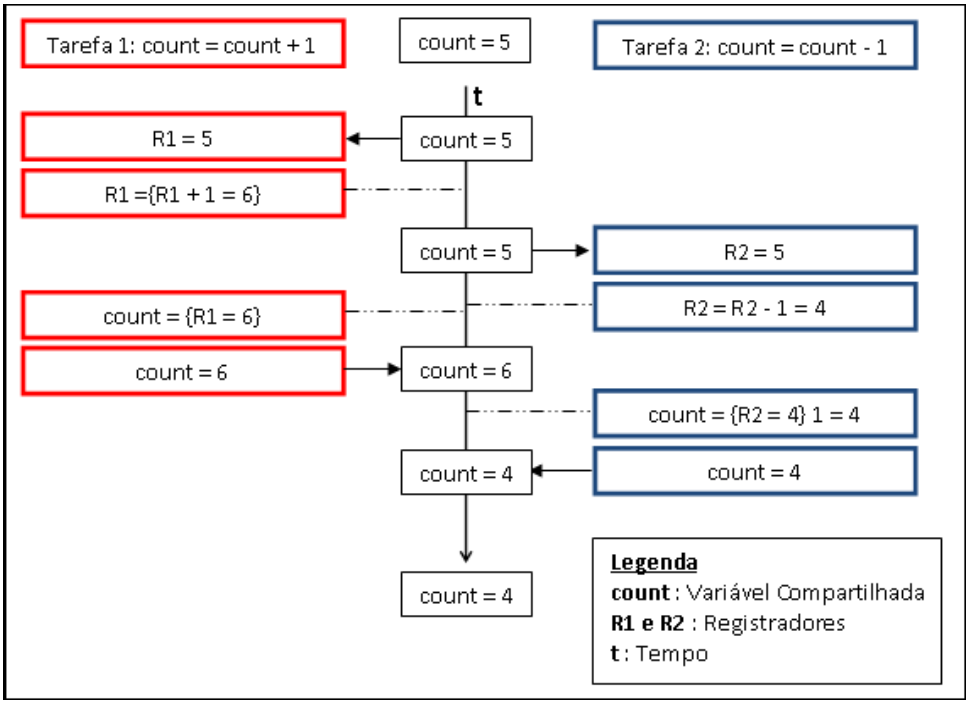

Figura 2.1: Condição de corrida (*race condition*)

para `count` era 5. Entretanto o resultado obtido foi 4. Ainda, se a ordem das escritas à variável `count` fosse invertida, o resultado seria 6. Esse estado incorreto ocorre porque duas tarefas manipulam de forma concorrente uma variável compartilhada.

É necessário evitar condições de corrida para garantir que o resultado de uma computação seja correto, independentemente do tempo (TOSCANI; OLIVEIRA; SILVA CARISSIMI, 2003). Além de incorreto, um programa com condições de corrida é de difícil depuração, pois a cada computação com os mesmos dados podem ser obtidos resultados diferentes. Isso contrasta com a programação sequencial onde, dada uma entrada de dados, os resultados na saída normalmente são os mesmos, facilitando a identificação e correção dos erros.

Há casos reais em que erros deste tipo causaram grandes problemas. Um exemplo clássico, bastante citado na literatura, é o caso da máquina de radioterapia Therac-25 (LEVESON; TURNER, 1993). Neste caso, uma condição de corrida fazia com que, às vezes, essa máquina administrasse dosagens milhares de vezes maior que o normal, resultando na morte ou em sérios danos para os pacientes (TOSCANI; OLIVEIRA; SILVA CARISSIMI, 2003).

Para evitar condições de corrida, operações de atualização não podem ser feitas si-

multaneamente por diferentes tarefas, tampouco operações de leitura podem ocorrer simultaneamente com atualizações, pois os dados lidos podem estar temporariamente inconsistentes (TOSCANI; OLIVEIRA; SILVA CARISSIMI, 2003). Então, devem ser tomadas precauções extras no acesso aos dados compartilhados com a finalidade garantir exclusão mútua, ou seja, permitir que somente uma tarefa de cada vez manipule um dado compartilhado.

Unindo os conceitos de seções críticas, condição de corrida e exclusão mútua, sabe-se que:

- Uma condição de corrida ocorre quando tarefas manipulam os dados compartilhados entre elas, em suas regiões críticas, de forma concomitante e desordenada.
- Para evitar o problema da condição de corrida, as seções críticas necessitam exclusão mútua.

2.1.2 Detectando condições de corrida

O problema para a identificação precisa de condições de corrida é considerado como NP-completo (NETZER; MILLER, 1992), ou seja, é muito difícil que se identifique precisamente todas as situações em que podem ocorrer problemas de concorrência. Porém, é possível desenvolver ferramentas que possibilitem a identificação de condições de corrida com um nível aceitável de precisão.

Existe um grande número de abordagens para a detecção de seções críticas e condições de corrida. Estas se utilizam, basicamente, de estratégias de análise dinâmica ou estática do programa. A análise dinâmica é feita em tempo de execução, geralmente acompanhando as linhas de execução do programa e registrando os eventos. Já na análise estática, a verificação é feita sobre o código-fonte do programa, sem a necessidade de executá-lo. A seguir serão apresentadas algumas dessas abordagens que estão relacionadas ao contexto deste trabalho.

2.1.2.1 Análise dinâmica

A detecção dinâmica de condição de corrida é feita através do monitoramento do programa durante sua execução (NETZER, 1991). As abordagens relacionadas à detecção dinâmica de condição de corrida são baseadas nos algoritmos: *happens-before*, *lockset* ou híbrido (utilizando os dois algoritmos) (LAMPORT, 1978; SAVAGE et al., 1997; DIN-

NING; SCHONBERG, 1991).

O algoritmo *happens-before* é baseado na relação *happens-before* de Lamport (LAMPORT, 1978), que é uma ordem parcial em todos os eventos. O algoritmo computa a relação de ordem parcial para determinar quais eventos acontecem antes de outro evento (eventos, nesse contexto, são todas as instruções, inclusive leitura/gravação e bloqueios) (PATIL; GEORGE, 2008). Se um par de acessos forem realizados por um par de *threads* em um local de memória e esses não sejam ordenados por esta relação, então eles são considerados envolvidos em uma condição de corrida (NAIK; AIKEN; WHALEY, 2006).

Os principais problemas do algoritmo *happens-before* é que ele é difícil de ser implementado de forma eficiente e, embora não produza falsos positivos, produz muitos falsos negativos (NAIK; AIKEN; WHALEY, 2006). Um exemplo de ferramenta baseada neste algoritmo é *Intel Thread Checker* (BANERJEE et al., 2006).

O algoritmo *lockset* detecta uma possível condição de corrida quando ocorre um par de acessos realizados por um par de *threads* a um local compartilhado de memória, sem que esses mantenham um bloqueio em comum. O principal problema com o algoritmo de detecção *lockset* é que ele produz muitos falsos positivos (NAIK; AIKEN; WHALEY, 2006), ou seja, nem todas as condições de corrida informadas por um algoritmo *lockset* são reais. É possível escrever um código livre de condições de corrida sem usar bloqueios, aplicando truques de programação inteligentes ou usando outros primitivos de sincronização como, por exemplo, sinalização (PATIL; GEORGE, 2008). Neste caso, o algoritmo geraria falsos positivos. Algumas ferramentas fazem uso deste algoritmo para detecção de condições de corrida, como por exemplo o programa *Eraser* (SAVAGE et al., 1997).

As abordagens que implementam algoritmos híbridos utilizam os algoritmos *happens-before* e *lockset* com a finalidade de aproveitar seus fatores positivos e reduzir os problemas relativos a estes algoritmos. Dinning and Schonberg (DINNING; SCHONBERG, 1991) foram os primeiros a explorar o uso de algoritmos híbridos. Desde então sugiram várias ferramentas, como por exemplo, *Visual Threads* (HARROW, 2000), *Hybrid Dynamic Data Race Detection* (O'CALLAHAN; CHOI, 2003), *MultiRace* (POZNIANSKY; SCHUSTER, 2003), *RaceTrack* (YU; RODEHEFFER; CHEN, 2005), *Sun Thread Analyzer* (SEREBRYANY; ISKHODZHANOV, 2009; MICROSYSTEMS, 2007) e *ThreadSanitizer* (SEREBRYANY; ISKHODZHANOV, 2009).

2.1.2.2 Análise estática

A detecção estática de condições de corrida examina o código-fonte do programa, analisando todos os caminhos de execução possíveis (NETZER, 1991).

Naik et al. (NAIK; AIKEN; WHALEY, 2006), propõe uma técnica para detecção estática de condições de corrida que tenta satisfazer alguns critérios, dos quais se destacam: precisão (evitar falsos positivos) e escalabilidade (ser capaz de lidar com programas grandes). O método proposto foi aplicado na ferramenta *chord* (jchord jchord, 2011) para análise de programas em Java.

Engler et al. apresenta *RacerX* (ENGLER; ASHCRAFT, 2003), uma ferramenta de detecção estática que utiliza análise sensível ao fluxo para detectar condições de corrida e *deadlocks*. A detecção é feita através da análise do controle de fluxo extraído do programa a ser analisado, utilizando um algoritmo de *lockset*. Os problemas detectados são classificados do maior ao de menor gravidade e apresentados ao usuário. *RacerX* foi concebido para encontrar erros em sistemas grandes e complexos. Segundo os autores, a ferramenta exige entre 2-14 minutos para analisar um sistema de 1,8 milhões de linhas.

Basupalli et al. (BASUPALLI et al., 2011) descreve a ferramenta de análise estática ompVerify. Seu objetivo é detecção de problemas de condição de corrida em laços de repetição de programas OpenMP. A ferramenta basicamente realiza uma busca por diretivas `omp parallel for` definidas de forma incorreta usando uma análise baseada no modelo poliédrico, e tem como saída mensagens de erro direcionadas ao usuário relativas a análise realizada. ompVerify destina-se a programas escritos na linguagem de programação C e é restrita a uma classe de programas chamada *Affine Control Loops* (ACLs). Foi desenvolvida com integração ao IDE Eclipse, utilizando os recursos do CDT/CODAN (eclipse.org eclipse.org, 2011) para análise estática e também de dois compiladores, GeCoS (INRIA, 2011) e AlphaZ para fazer a análise poliédrica e detectar erros.

Embora a detecção estática de condições de corrida tenha conseguido grandes avanços, as ferramentas de detecção dinâmica ainda são predominantes. Detectores dinâmicos desfrutam de maior precisão e escalabilidade, entretanto são difíceis de usar em programas no início do desenvolvimento (NAIK; AIKEN; WHALEY, 2006). Como a análise dinâmica é feita através da execução do programa, são necessários códigos e dados suficientes para realização de testes. Já a análise estática pode ser feita em vários estágios do

desenvolvimento do programa, inclusive no início.

As abordagens apresentadas procuram detectar condições de corrida em programas já paralelizados, caracterizando-as como detectores de problemas em um programa paralelo. A proposta deste trabalho assemelha-se a estas abordagens no fato de identificar possíveis problemas de concorrência, entretanto o objetivo é diferente, pois o trabalho não visa buscar por defeitos em um código paralelo e sim busca auxiliar na transformação de um código sequencial em um código paralelo, identificando variáveis que podem vir a ser envolvidas em uma condição de corrida durante a execução paralela do código.

2.2 Transformação de código sequencial em código paralelo

A transformação de código sequencial em código paralelo pode ser feita de forma manual, automatizada ou semi-automatizada. A transformação feita de forma manual não é uma tarefa trivial. Ela exige tempo, atenção e certo nível de conhecimento do programador, elevando o risco do código paralelizado conter erros. As formas automatizadas e semi-automatizadas de transformação de código auxiliam e dão maior segurança a este processo. Tais abordagens são desenvolvidas em compiladores e ferramentas para paralelização de código.

Nesta seção, apresenta-se inicialmente o padrão OpenMP e sua utilização na paralelização manual de código. Em seguida, discute-se a transformação de forma automatizada ou semi-automatizada, incluindo o papel da refatoração neste contexto.

2.2.1 Paralelização com OpenMP

OpenMP (*Open Multi-Processing*) (OPENMP, 2011) é uma API (*Application Program Interface*) para a programação paralela com múltiplas *threads* em arquiteturas de memória compartilhada. Pode ser utilizada nas linguagens Fortran (77, 90, 95), C e C++ (ARB, 2008). Sua primeira versão, OpenMP 1.0, foi publicada no ano de 1997, e atualmente encontra-se na versão 3.1 publicada em 2011.

Trata-se de um modelo de programação portável e escalável, que proporciona aos programadores uma interface simples e flexível para o desenvolvimento de aplicações paralelas. É constituída por um conjunto de diretivas de compilador, rotinas de biblioteca e variáveis de ambiente que influenciam o comportamento do tempo de execução.

Os compiladores traduzem as diretivas para as primitivas do sistema operacional ba-

seadas em *threads*. Visto que essas diretivas podem ser tratadas como comentários, o programa paralelizado com OpenMP pode também ser executado sequencialmente, caso o compilador não tenha suporte a esta ferramenta. Sua aplicação pode ser feita de forma incremental, combinando código sequencial e paralelo num mesmo programa (TOMITA, 2004).

A paralelização com OpenMP é realizada com múltiplas *threads*, em um modelo *fork-join* de execução (ARB, 2008). O programa inicia com uma única *thread*, chamada *thread* mestre, que executa sozinha até encontrar uma região paralela (marcada pela diretiva `parallel`). Nesse ponto do programa, é criado um grupo de *threads* que executarão concorrentemente o código dentro dessa região. Quando todas as *threads* terminarem de executar o código da região paralelizada, elas são sincronizadas e, com exceção da *thread* mestre, são destruídas. Dessa forma, a *thread* mestre continua a execução sequencial do programa até encontrar uma nova região paralelizada ou terminar a execução do programa.

Dentro de uma região paralela, as variáveis podem ser **privadas** (cláusula `private`) ou **compartilhadas** (cláusula `shared`). Todas as *threads* veem a mesma cópia das variáveis compartilhadas e veem a sua cópia das variáveis privadas, que não são visíveis pelas outras *threads* (TOMITA, 2004). Portanto, é necessário adotar alguns cuidados na manipulação das variáveis nas regiões paralelas, como por exemplo, adicionando diretivas de sincronização da execução e definindo corretamente as varáveis privadas e compartilhadas, evitando gerar resultados incoerentes.

A figura 2.2 apresenta um simples exemplo contendo um trecho de código Fortran com diretivas OpenMP para paralelização de um laço de repetição. Neste exemplo define-se que o laço de repetição será executado em paralelo pelas *threads*, sendo que cada *thread* ficará encarregada por executar uma parte das iterações do laço. Por exemplo, se o controlador do laço `i`, que inicia inicia em um (1), terminará em cem (100) e o trecho será executado por quatro (4) *threads*, cada *thread* será responsável por vinte e cinco (25) iterações do laço. Neste exemplo, é definido também quais variáveis serão privadas e quais serão compartilhadas respectivamente pelas diretivas `PRIVATE` e `SHARED`.

Embora as construções OpenMP sejam simples e facilitem a transformação de código sequencial em código paralelo, tal transformação, se feita manualmente, é passível de conter erros que muitas vezes são difíceis de serem identificados pelo programador. As-

```
....

!$OMP PARALLEL DO DEFAULT(NONE)       &
!$OMP SHARED(m,n,a,b,c) PRIVATE(i,j)
      do i = 1, m
         a(i) = 0.0
         do j = 1, n
            a(i) = a(i) + b(i,j)*c(j)
         end do
      end do
!$OMP END PARALLEL DO

...
```

Figura 2.2: Exemplo de paralelização de laço de repetição em OpenMP para Fortran (CHAPMAN; JOST; PAS, 2007)

sim, fica evidente a utilidade de ferramentas que auxilie no processo de transformação do código, como é o caso do analisador proposto neste trabalho.

2.2.2 Transformação automática ou semi-automática

Existem muitas abordagens sobre a pesquisa e desenvolvimento de ferramentas que realizam ou auxiliam na transformação de código sequencial em código paralelo. Esses estudos aumentaram a partir da década de 90 e continuam sendo temas de pesquisa atualmente. A maioria dos trabalhos esta voltada para a otimização de laços de repetição, variando entre técnicas de otimização, *frameworks* de paralelização (reunindo compiladores, algoritmos e ferramentas) e arquiteturas computacionais a que se destinam.

Os compiladores analisam o código fonte e identificam os pontos de paralelização, que geralmente tratam-se de laços de repetição. Um exemplo desses compiladores é o Oxygen (RÜHL, 1992), que gera código paralelo a partir de códigos em Fortran 77 destinados a arquiteturas paralelas com memória distribuída. Outros exemplos de compiladores paralelizadores são o SUIF (WILSON et al., 1994), o OSCAR (KIMURA et al., 2005) e o PARADIGM (BANERJEE et al., 1995).

Os compiladores paralelizadores possuem fatores positivos, como a facilidade de paralelização de código e a não exigência de um conhecimento avançado do usuário. Entretanto, fatores negativos como a pouca flexibilidade, relação muito próxima com o tipo de arquitetura e o ganho de desempenho muitas vezes pouco satisfatório, acabam não satisfazendo as necessidades dos programadores. Já as ferramentas de auxílio à paralelização são mais flexíveis, e é nessa linha que segue a abordagem deste trabalho.

Assim como em compiladores, há muitos trabalhos de pesquisa relacionados ao desen-

volvimento de ferramentas de paralelização. Mitra et al. apresentam uma ferramenta interativa para paralelização de código legado denominada ParAgent (MITRA et al., 2000). A abordagem requer alto nível de conhecimento de paralelização, com o usuário oferecendo um roteiro para a paralelização. É destinada para aplicações científicas que utilizam o método de diferenças finitas como, por exemplo, aplicações de previsão numérica do tempo, utilizando-se de características específicas desse método para orientar o processo de paralelização automática. Essa ferramenta recebe como entrada um programa sequencial em Fortran 77 e tem como saída uma programa paralelo, também em Fortran 77, com primitivas para a troca de mensagens em sistemas de memória distribuída.

Apesar de realizar a paralelização de códigos sequenciais de forma automatizada e utilizar análise estática do código, ParAgent diferencia-se da abordagem deste trabalho por ser voltada para uma classe específica de programas e sistemas de memória distribuída. Este trabalho busca auxiliar na paralelização de programas sequenciais, independentemente da área de aplicação, e preocupa-se com as características dos sistemas de memória compartilhada.

Cronus (ANDRONIKOS et al., 2008) é outro exemplo de ferramenta. Ela realiza a paralelização de programas em C, focando o paralelismo em laços de repetição aninhados. Cronus recebe como entrada um arquivo C com várias diretivas para a ferramenta e o converte para um programa C paralelo baseado em MPI, que adicionalmente usa rotinas de biblioteca do Cronus para lidar com tarefas que satisfazem as dependências entre iterações dos laços de repetição. A abordagem usa o algoritmo QuickHull (BARBER; DOBKIN; HUHDANPAA, 1996) para analisar os laços e análise dinâmica para identificar as iterações e gerar o código paralelo. Assim como ParAgent, Cronus destina-se à uma classe restrita de programas, tendo como foco a otimização de laços aninhados, gerando código paralelo com troca de mensagens.

Outra ferramenta, a CAPO (*Computer-Aided Parallelizer and Optimizer*) (JIN; FRUMKIN; YAN, 2000), desenvolvida em um projeto da NASA (*National Aeronautics and Space Administration*), traz uma abordagem muito próxima à proposta deste trabalho. Nela automatiza-se a inserção de diretivas OpenMP para facilitar o processamento paralelo em máquinas memória compartilhada. CAPO é totalmente integrada com a CAP-Tools (IEROTHEOU et al., 1996) que, por sua vez, é responsável por fazer a análise de dependência dos dados do programa.

CAPO recebe como entrada programas sequenciais em Fortran, realiza análise de dependência inter-procedural de dados e gera diretivas OpenMP. A geração de diretivas de compilador é feita em três etapas:

- Identificação de laços paralelos no nível mais externo;
- Construção de regiões paralelas em torno de laços paralelos e otimização de regiões paralelas;
- Inserção das diretivas com identificação automática de redução, privatização, indução e variáveis compartilhadas.

Embora diretivas OpenMP sejam geradas automaticamente, a interação do usuário com a ferramenta é importante para a produção de bons códigos paralelos. Para tanto, uma interface gráfica abrangente é oferecida para o usuário interagir com o processo de paralelização.

Das ferramentas automáticas de conversão de código serial em paralelo, como compiladores por exemplo, a maioria não requer grande conhecimento por parte do programador. Entretanto, o seu bom desempenho é muitas vezes restrito a uma classe de programas, que normalmente envolvem operações de matrizes densas e cálculos *stencil*, desmotivando a sua utilização por parte dos programadores (DIG, 2011).

Ultimamente o conceito de refatoração vêm ganhando força para agir nesse campo. Ferramentas de refatoração seguem uma linha diferente dos paralelizadores, onde servem de apoio ao programador. Nesse tipo de abordagem, o programador segue com o controle das alterações utilizando seu conhecimento sobre o domínio, definindo o que deve ser alterado e onde devem ser feitas as transformações. Cabe a ferramenta de refatoração executar o trabalho tedioso, analisando e efetuando as alterações de código com maior rapidez e segurança. Dessa forma, une-se o conhecimento do programador e a eficiência de uma ferramenta computacional (DIG, 2011).

2.2.3 Refatoração

Refatoração é uma alteração feita na estrutura interna do *software* para torná-lo mais fácil de ser entendido e menos custoso de ser modificado, sem alterar seu comportamento observável (OPDYKE, 1992; FOWLER et al., 1999; TOURWÉ; MENS, 2004). Esse processo normalmente envolve a remoção de código duplicado, simplificação de lógica

condicional e melhora de legibilidade de código (KERIEVSKY, 2004). Portanto, a ação de refatoração consiste em aplicar modificações na estrutura interna do programa, mais especificamente em seu código, sem que isso altere suas funcionalidades. O programa refatorado deve continuar a produzir os mesmos resultados de sua versão original.

O termo refatoração foi originalmente introduzido como uma variante da reestruturação de *software* (OPDYKE, 1992; ARNOLD, 1986; GRISWOLD; NOTKIN, 1993). Considerando o contexto da evolução de *software*, refatoração e reestruturação convergem quanto ao objetivo de melhorar a qualidade, principalmente sobres aspectos como: extensibilidade, modularidade, reusabilidade, complexidade e eficiência do *software* (TOURWÉ; MENS, 2004).

A primeira ferramenta para automatização de técnicas de refatoração desenvolvida foi a *Refactoring Browser* (ROBERTS; BRANT; JOHNSON, 1996, 1997), uma ferramenta para código Smalltalk que introduzia pela primeira vez a automação de refatorações primitivas.

Na última década, as ferramentas de refatoração obtiveram grande sucesso na programação sequencial (DIG, 2011). A facilidade de utilização por meio de IDEs interativas tornou natural o processo de refatoração que antes era deixado de lado em virtude de ser uma tarefa difícil e custosa. A expectativa é de que esse mesmo sucesso seja obtido na refatoração de código sequencial em código paralelo. Com a grande disseminação de arquiteturas computacionais multiprocessadas, esse tipo de refatoração passa a ser uma necessidade para evolução dos programas computacionais e, como não é uma tarefa trivial, as ferramentas automatizadas serão de grande importância.

Nesse contexto, visando explorar esta necessidade emergente, ferramentas de refatoração estão sendo propostas nos últimos anos. Em Dig et al. (DIG; MARRERO; ERNST, 2009), é apresentada ferramenta CONCURRENCER, que permite a refatoração de código sequencial em código paralelo em linguagem Java, transformando o código-fonte para que este utilize-se da biblioteca Java 5 *java.util.concurrent (j.u.c.)* de suporte a programas concorrentes. A ferramenta é integrada com o mecanismo de refatoração do Eclipse.

Outra ferramenta de refatoração de código sequencial para código paralelo destinada a linguagem java é o ReLooper (DIG et al., 2009). Esta ferramenta explora o uso de uma estrutura de dados chamada *ParallelArray* que permite operações paralelas sobre seus elementos. ReLooper propõe-se a transformar um *array* selecionado pelo programador para

um *ParallelArray*, analisando se as iterações de laços de repetição que manipulam este *array* são seguras para execução paralela e então faz a substituição destes laços com as operações paralelas equivalentes. Se, durante a análise do laço de repetição, é verificada a possibilidade de ocorrem problemas, como por exemplo condição de corrida, estes são relatados ao usuário, que por sua vez, pode corrigi-los ou ignorá-los seguindo com a refatoração, mantendo assim, o controle das alterações sob responsabilidade do programador.

Reentrancer (WLOKA; SRIDHARAN; TIP, 2009) é mais uma ferramenta de refatoração criada recentemente. Seu objetivo é refatorar programas reentrantes para permitir o paralelismo, para isso, a ferramenta substitui o estado global mutável pela *thread* local, desta forma cada *thread* possui uma cópia da variável global. Reentrancer foi desenvolvida pela IBM no contexto do Eclipse JDT.

Este trabalho faz uso dos conceitos de refatoração para atuar na transformação de código sequencial em paralelo, auxiliando o programador nesse processo. Assim como as abordagens apresentadas que usam refatoração (DIG; MARRERO; ERNST, 2009; DIG et al., 2009), este trabalho busca unir o conhecimento do programador como a agilidade e a segurança de uma ferramenta automatizada, além de também fazer uso de uma IDE (Eclipse) para a implementação da ferramenta. Todavia, embora o algoritmo seja implementado usando infraestrutura de refatoração, seu resultado não é uma refatoração de código propriamente dita, mas sim um subsídio para auxiliar o programador na refatoração. O algoritmo proposto e sua implementação através de uma refatoração serão apresentados no capítulo a seguir.

3 DESENVOLVIMENTO

O objetivo deste trabalho é auxiliar na refatoração de um código sequencial para paralelo, realizando uma análise de acessos concorrentes a dados de forma automatizada. Para isso, foi desenvolvido um algoritmo que faz a análise do código-fonte de forma estática na procura de variáveis que podem causar problemas de condição de corrida em uma execução paralela do programa. Este algoritmo foi implementado usando um *framework* que fornece recursos para manipular a árvore sintática de programas Fortran.

3.1 Requisitos

3.1.1 Entrada e saída

O algoritmo de análise recebe como **entrada** um trecho de código que o programador deseja paralelizar. Com base nesse trecho, são analisados os acessos às variáveis na busca por situações que podem vir a caracterizar um problema de corrida, caso o trecho de código seja executado por várias *threads*. Após essa análise, o algoritmo terá como **saída** as variáveis críticas, ou seja, as variáveis que poderão estar envolvidas em condições de corrida. Além disso, também terá como saída a localização dessas variáveis no código, com a finalidade de facilitar a visualização dos resultados por parte do programador.

O funcionamento geral do algoritmo é o seguinte: dado um trecho de código de entrada, verifica-se cada uma das variáveis nele contidas, cujo valor esteja sendo alterado. Além disso, é verificado se a mesma variável está sendo lida. A verificação de leitura consiste basicamente em procurar em outras partes do trecho e/ou em procedimentos referenciados pelo mesmo, se a variável em análise está sendo utilizada em alguma operação de leitura além da sua atribuição. Nessa situação, além de seu valor estar sendo escrito, ele também estará sendo lido durante a execução desse trecho de código, sendo esse fato um indício de que ela pode vir a ser envolvida em uma condição de corrida durante a

execução paralela do trecho.

A figura 3.1 apresenta um exemplo simples para a verificação de escrita e leitura. Do lado esquerdo da imagem está o código selecionado a ser analisado pelo algoritmo e do lado direito o resultado da verificação. Neste exemplo, o algoritmo deve detectar as variáveis **"*i*"**, **"*a*"** e **"*b*"** como possíveis variáveis críticas. Elas devem ser detectadas por possuírem operações de escrita e leitura sobre seus dados, como está destacado no lado direito (resultado) da figura. Já as variáveis **"*N*"** e **"*c*"** não representam perigo de corrida, pois possuem apenas a operação de leitura sobre seus dados.

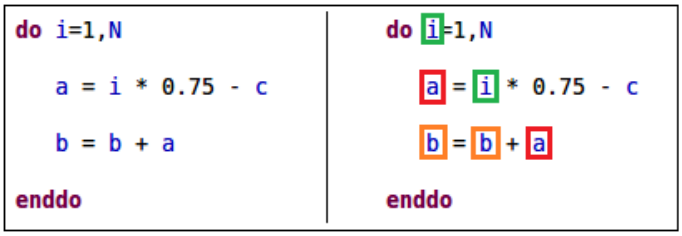

Figura 3.1: Exemplo de verificação

Durante a verificação, não basta apenas localizar no código a ocorrência do nome da variável, pois, além de poder ser representada por nomes diferentes em funções distintas, o nome da variável pode ser repetido em vários locais do código, porém com um contexto diferente. Ou seja, a mesma cadeia de caracteres que representa um determinado dado pode estar representando um elemento diferente em outro local do código. Sendo assim, para possibilitar tal verificação é necessário o auxílio de um analisador sintático que possibilite ao algoritmo obter e atuar sobre a **árvore sintática** do código.

Tendo como entrada um trecho de código para análise e o auxílio de um analisador sintático para obter e manipular a árvore sintática do código, viabiliza-se a implementação do algoritmo proposto.

3.1.2 Árvore sintática

Boa parte das funções contidas no algoritmo de detecção são providas pelo analisador sintático para tratar das questões relacionadas a manipulação da árvore sintática. Sendo elas:

- ***obterTrechoDeCodigoSelecionado***: retorna o código selecionado no editor;
- ***obterListaDeNos***: retorna a lista de nós referente ao trecho de código selecionado;

- ***obterNumeroDeNos***: retorna o número de nós contidos na lista de nós;
- ***obterNo***: retorna o nó de uma determinada posição da lista;
- ***obterVariavelAtribuida***: retorna qual é a variável que está sendo alterada;
- ***obterIndiceDoLaco***: retorna a variável que representa o índice do laço de repetição;
- ***obterSubRotina***: retorna a lista de nós da implementação da sub-rotina;
- ***obterNumeroParametros***: retorna o número de parâmetros da sub-rotina;
- ***obterParametro***: retorna o símbolo referente ao parâmetro;
- ***obterSimbolosAlterados***: retorna uma lista de símbolos que estão sendo alterados no nó;
- ***obterSimbolosLidos***: retorna uma lista de símbolos que estão sendo lidos no nó;
- ***obterSimbolo***: retorna um símbolo;
- ***obterReferencia***: retorna a referência de um símbolo, utilizada para verificar se dois ou mais símbolos representam o mesmo dado e onde está a definição deste símbolo;
- ***IgualAtribuicao, IgualLacoRepeticao,IgualChamadaDeProcedimento, IgualGlobal***: verifica se um determinado nó é uma operação de atribuição, um laço de repetição, uma chamada de procedimento ou se o símbolo é uma variável global, retornando verdadeiro ou falso.

3.2 Algoritmo de análise

O algoritmo de análise divide-se em três partes, conforme mostra o diagrama de sequência apresentado na figura 3.2. O **analisador** recebe o trecho de código de entrada, percorre sua árvore sintática indicando as variáveis e instruções a serem analisadas pelas outras etapas do algoritmo e armazena os resultados dessas análises.

A análise de **variável crítica** verifica os acessos a uma variável, indicando se eles caracterizam a possibilidade dessa variável vir ser envolvida em uma condição de corrida em uma execução paralela. A análise de **chamada de procedimento**, por sua vez, aplica a análise de variável crítica no código de procedimentos que porventura sejam invocados

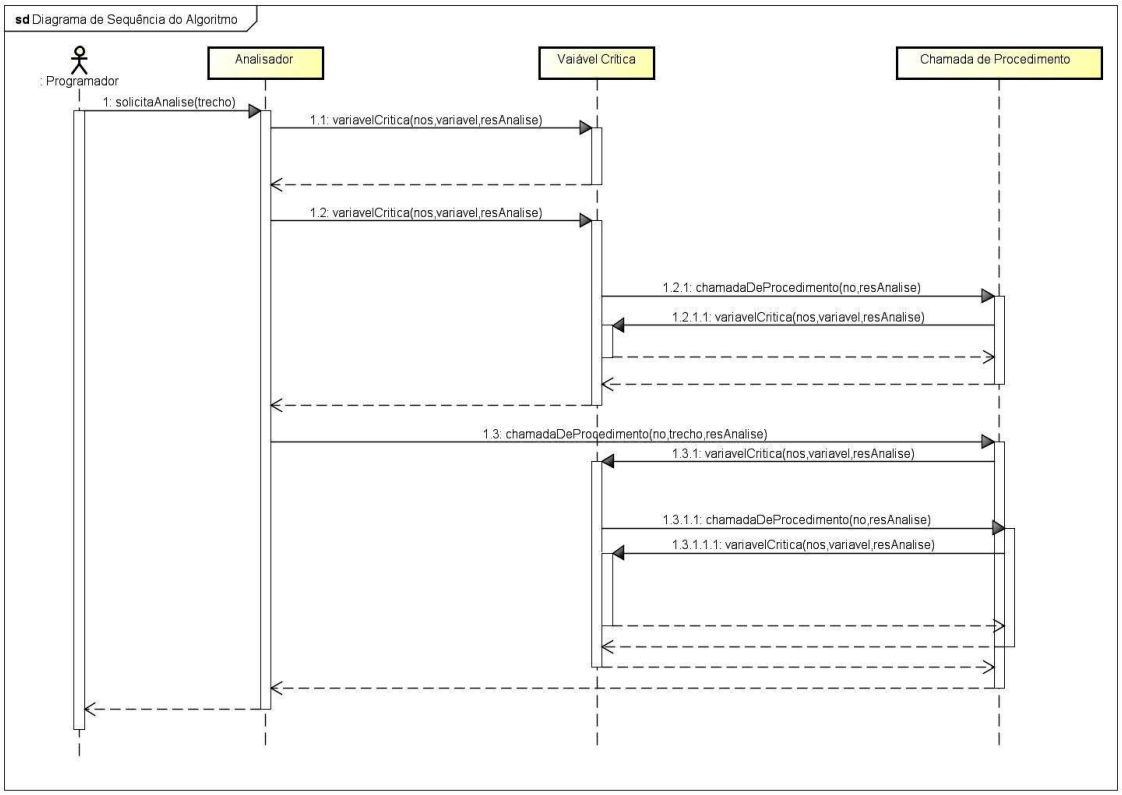

Figura 3.2: Diagrama do algoritmo

no código de entrada. Isso é necessário porque um procedimento pode ler e alterar alguma variável do trecho selecionado. A interação entre estas partes do algoritmo pode ser vista na descrição individual de cada uma delas a seguir.

3.2.1 Algoritmo principal do analisador

O algoritmo principal do analisador (Algorithm 1) trabalha primeiramente com a entrada de dados do algoritmo, recebendo o trecho de código selecionado e também a sua árvore sintática através de um analisador sintático. Feito isso, a árvore é percorrida e seus nós vão sendo analisados com o auxílio de funções disponibilizadas pelo analisador sintático e, após a verificação, as variáveis detectadas são adicionadas ao resultado. Ao percorrer a árvore referente ao trecho são verificados dois tipos de nós:

- **Operação de atribuição**: obtêm-se a variável que está sendo alterada através de uma função provida pelo analisador sintático e verifica se é uma variável crítica através do método `variavelCritica`;
- **Chamada de procedimento**: invoca o método `chamadaDeProcedimento` passando o nó da chamada.

Algorithm 1 analise
Require: trecho

$no : NO$
$variavel : SIMBOLO$
$resAnalise : MATRIZ$
$nos : ARVORE\ de\ NOS$
$i : VALOR$

$nos \leftarrow trecho.obterArvoreDeNos$
for $i \leftarrow 1 \rightarrow i = nos.obterNumeroDeNos$ **do**
  $no \leftarrow nos.obterNo(i)$
  if $no.IgualAtribuicao$ **then**
    $variavel \leftarrow no.obterVariavelAtribuida$
    $resAnalise \leftarrow variavelCritica(nos, variavel, \textbf{false}, resAnalise)$
  else if $no.IgualChamadaDeProcedimento$ **then**
    $resAnalise \leftarrow chamadaDeProcedimento(no, nos, NULO, \textbf{true}, \textbf{true}, resAnalise)$
  end if
end for
print $resAnalise$

3.2.2 Verificação de variável crítica

No método `variavelCritica` (Algorithm 2) é verificado se há possibilidade de uma determinada variável ser envolvida em uma condição de corrida. Para isso, é verificado se a variável está sendo escrita e lida durante a execução do trecho. Nesse método, é verificada uma situação por chamada de acordo com o parâmetro `verificarAtribuicao`. Se seu conteúdo for verdadeiro verificará se a variável está sendo escrita; se for falso, verificará se a variável está sendo atribuída. Se, ao verificar uma leitura da variável o resultado desse método for verdadeiro, a variável analisada é adicionada à lista de resultados na `analise` ou no método `chamadaDeProcedimento` dependendo de onde foi invocado o método `variavelCritica`.

Além dessa verificação, nesse método são verificadas as variáveis globais alteradas e lidas em um determinado trecho de código. Essas variáveis são adicionadas em duas listas globais, a lista de variáveis alteradas e lista de variáveis lidas. Posteriormente é feito um cruzamento entre as listas: caso uma determinada variável esteja na lista de variáveis alteradas e também na lista de variáveis lidas, ela representa risco de corrida. Sendo assim, esta é adicionada ao resultado da análise. Esta verificação diferenciada é necessária pois as variáveis podem ser manipuladas em partes distintas do código como,

Algorithm 2 variavelCritica

Require: codigo:ARVORE de NOS, variavel:SIMBOLO, verifAtribuicao:LOGICO, resAnalise:MATRIZ

$varGlobaisAlteradas, varGlobaisLidas, simbolos : LISTA\ de\ SIMBOLOS$
$s : SIMBOLO$
$i, j : VALOR$

for $i \leftarrow 1 \rightarrow codigo.obterNumeroDeNos$ **do**
  if $codigo.obterNo[i].IgualChamadaDeProcedimento$ **then**
    $resAnalise \leftarrow chamadaDeProcedimento(codigo.obterNo[i], codigo, variavel,$
    false$, verifAtribuicao, resAnalise)$
  end if

  $simbolos \leftarrow codigo.obterNo[i].obterSimbolosAlterados$
  for $j \leftarrow 1 \rightarrow simbolos.obterNumeroDeSimbolos$ **do**
    $simbolo \leftarrow simbolos.obterSimbolo[j]$
    if $simbolo.obterReferencia.IgualGlobal$ **then**
      $varGlobaisAlteradas.add(simbolo.obterReferencia)$
      if $varGlobaisLidas.contem(simbolo.obterReferencia)$ **then**
        $resAnalise.add(simbolo.obterReferencia, simbolo.obterLocalizacao)$
      end if
    end if
    if $verifAtribuicao$ **then**
      if $simbolo.obterReferencia = variavel.obterReferencia$ **then**
        $resAnalise.add(simbolo.obterReferencia, simbolo.obterLocalizacao)$
      end if
    end if
  end for

  $simbolos \leftarrow codigo.obterNo[i].obterSimbolosLidos$
  for $j \leftarrow 1 \rightarrow simbolos.obterNumeroDeSimbolos$ **do**
    $simbolo \leftarrow simbolos.obterSimbolo[j]$
    if $simbolo.obterReferencia.IgualGlobal$ **then**
      $varGlobaisLidas.add(simbolo.obterReferencia)$
      if $varGlobaisAlteradas.contem(simbolo.obterReferencia)$ **then**
        $resAnalise.add(simbolo.obterReferencia, simbolo.obterLocalizacao)$
      end if
    end if
    if not $verifAtribuicao$ **then**
      if $simbolo.obterReferencia = variavel.obterReferencia$ **then**
        $resAnalise.add(simbolo.obterReferencia, simbolo.obterLocalizacao)$
      end if
    end if
  end for
end for

return $resAnalise$

por exemplo, sub-rotinas diferentes utilizadas na execução do trecho em análise.

3.2.3 Verificação de chamada de procedimento

No método `chamadaDeProcedimento`, Algorithm 3, são obtidos os parâmetros da sub-rotina e os mesmos são verificados pelo método `variavelCritica`. Caso não haja parâmetros, são verificadas apenas as variáveis globais contidas na sub-rotina. Entretanto, antes de fazer a verificação, é necessário identificar de onde partiu a chamada para o método `chamadaDeProcedimento`. Esse controle é feito com o auxílio da variável lógica `primeiraVerificacao`, caso seu valor seja verdadeiro, indica que a chamada é referente a parte inicial do algoritmo, ou seja, (`analise`). Caso contrário, indica que a chamada é referente ao método `variavelCritica`.

Se a chamada for referente a `analise`, é necessário fazer a verificação de escrita e de leitura sobre as variáveis a serem analisadas. Portanto, o método `variavelCritica` é chamado duas vezes sendo modificado apenas o parâmetro `verificarAtribuicao`. Já se a chamada é referente ao método `variavelCritica`, é feita uma única verificação passando como parâmetro a variável `verificarAtribuicao` recebida pelo método.

As chamadas realizadas a partir do método `variavelCritica` direcionadas ao método `chamadaDeProcedimento` que, por sua vez, chamará novamente o método `variavelCritica`, dão ao algoritmo a característica de recursividade. Esta relação pode ser vista na figura 3.2.

3.2.4 Limitações do algoritmo

O algoritmo apresenta também algumas limitações. Como pode ser visto na sua descrição, todas as variáveis são tratadas da mesma forma, independente de sua estrutura. Desse modo, estruturas de dados como vetores, matrizes e tipos estruturados serão tratados da mesma forma que uma simples variável numérica, por exemplo. Essa situação diminui consideravelmente a precisão na detecção de possíveis corridas relativas a estas estruturas. Nesse caso, para se ter uma verificação eficiente de acesso a dados na busca por possíveis corridas é necessário analisar seus índices e fluxo de dados, tarefa essa de grande dificuldade na análise estática, pois a análise teria que se aproximar muito da execução do programa.

Outra limitação está relacionada a passagem de parâmetros para as sub-rotinas, não

Algorithm 3 chamadaDeProcedimento

Require: chamada:NO, trecho: ARVORE de NOS, variavel:SIMBOLO, primeiraVerificacao:LOGICO, verifAtribuicao:LOGICO, resAnalise: MATRIZ

$parChamada, parSubrotina : LISTA\ de\ SIMBOLOS$
$subrotina : ARVORE\ de\ NOS$
$i : VALOR$
$auxAnalise : MATRIZ$

$subrotina \leftarrow chamada.obterSubRotina$
for $i \leftarrow 1 \rightarrow chamada.obterNumeroParametros$ **do**
  if $(chamada.obterParametro[i] = variavel)$**or**$(variavel = NULO)$ **then**
    $parChamada.adiciona(chamada.obterParametro[i])$
    $parSubrotina.adiciona(subrotina.obterParametro[i])$
  end if
end for

if $parChamada.obterNumeroDeElementos = 0$ **then**
  $resAnalise \leftarrow variavelCritica(rotina, NULO, verifAtribuicao, resAnalise)$
else
  for $i \leftarrow 1 \rightarrow parChamada.obterNumeroDeElementos$ **do**
    if $primeiraVerificacao$ **then**
      $auxAnalise \leftarrow variavelCritica(rotina, parSubrotina[i], verifAtribuicao, resAnalise)$
      if $resAnalise.obterTamanho > auxAnalise.obterTamanho$ **then**
        $resAnalise \leftarrow variavelCritica(trecho, parChamada[i],$ **false**$)$
        $resAnalise \leftarrow variavelCritica(rotina, parSubrotina[i],$ **false**$)$
      end if
    else
      $resAnalise \leftarrow variavelCritica(rotina, parSubrotina[i], verifAtribuicao, resAnalise)$
    end if
  end for
end if

return $resAnalise$

sendo considerado pelo algoritmo se a passagem de parâmetro de uma determinada variável em análise é somente para leitura ou também para escrita. Essa definição da passagem de parâmetro dependerá da linguagem de programação, podendo o algoritmo ser adequado durante a sua implementação para uma determinada linguagem.

3.3 Implementação

O algoritmo de análise apresentado na seção anterior foi implementado tendo como alvo programas em linguagem Fortran. Para isso, utilizou-se a ferramenta Photran, que possui um analisador sintático para a linguagem em questão e possibilita a instanciação do algoritmo como uma etapa inicial de uma refatoração para código Fortran. A funcionalidade implementada integra-se ao IDE Eclipse (menu refatoração) sob o nome *"Detect Possible Critical Sections"*. Na sequência, apresenta-se a ferramenta Photran e a implementação do algoritmo proposto.

3.3.1 Photran

Photran (ECLIPSE.ORG, 2011) é um ambiente integrado de desenvolvimento (IDE) para código Fortran baseado na plataforma Eclipse (BEATON; RIVIERES, 2006), sendo uma extensão do *plugin* CDT (*C/C++ Development Tools*) (eclipse.org eclipse.org, 2011). O Eclipse é um IDE de desenvolvimento que tem suporte para diversas linguagens de programação, como C, C++, Java, e recentemente Fortran, com o uso do *plugin* Photran.

Mantido pelo projeto Eclipse como um de seus *plugins* oficiais, Photran é desenvolvido em linguagem Java e é composto de *plugins* e de recursos (*features*). *Plugins* adicionam funcionalidades ao Eclipse, enquanto os recursos são unidades de desenvolvimento (diversos *plugins* são empacotados em recursos, que são distribuídos aos usuários). Arquiteturalmente, o projeto Photran divide-se em subprojetos onde cada um se constitui de um *plugin* ou um recurso.

A ideia central do projeto é de disponibilizar um *framework* que possibilite o rápido desenvolvimento de ações de refatoração para código Fortran reutilizando-se de boa parte da infra-estrutura provida pelo Eclipse (DE, 2004).

As automações de refatoração no *plugin* Photran são aplicadas através da manipulação de ASTs (*Abstract Syntax Tree*). Uma árvore sintática abstrata é uma estrutura para representação do código-fonte do programa. Contitui-se de uma raiz da qual são deriva-

dos vários nós que por sua vez compõe-se de outros nós. O último nível dessa árvore geralmente representa os elementos da gramática da linguagem de programação (*tokens* - comandos, expressões, operadores, etc.) (JONES, 2003). Cada nó é classificado de acordo com seu funcionamento e suas ações. A figura 3.3 apresenta um exemplo simples, onde um nó do tipo declaração é derivado por outros nós que representam o tipo e o nome da variável declarada.

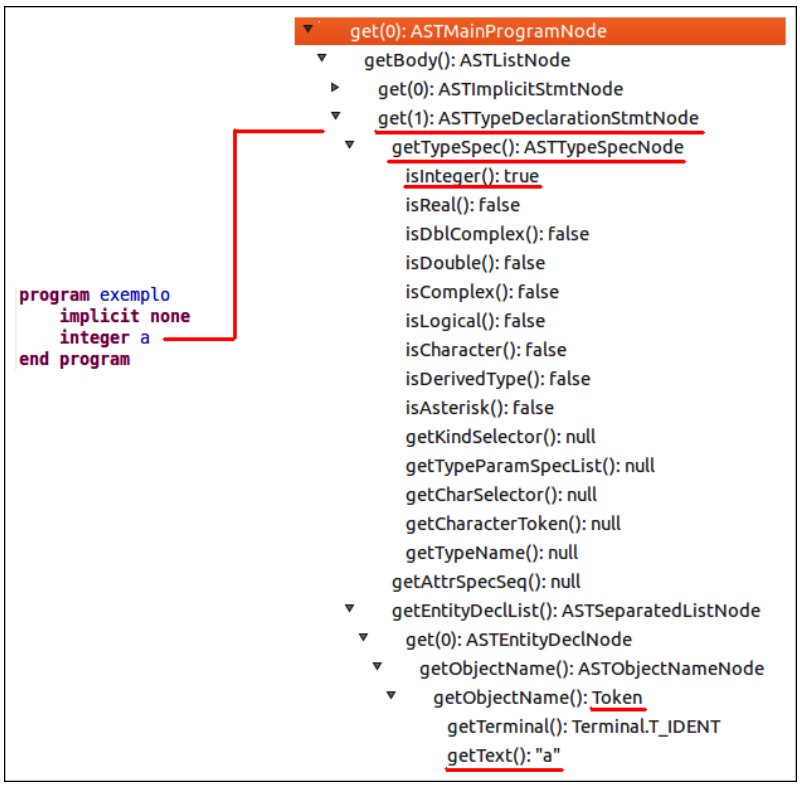

Figura 3.3: AST - *Abstract Syntax Tree*

A construção da AST é possível em função do analisador sintático para a linguagem Fortran utilizado pelo *plugin* Photran. O analisador sintático é responsável por validar a estrutura de um código fonte e gerar uma sequência de derivação, ou seja, uma árvore sintática abstrata (AST). Caso o código fonte não possa ser decomposto segundo a estrutura gramatical, o processo de análise sintática acusa um erro de sintaxe.

A AST possibilita ao desenvolvedor navegar pela estrutura do código-fonte, detectando correlações entre seus nós e identificando oportunidades de refatoração, sendo peça chave para a automatização de ações de refatoração (OVERBEY; JOHNSON, 2009). A ação de refatoração em geral introduz modificações na estrutura da AST (incluindo, excluindo ou alterando o posicionamento dos nós).

Uma refatoração no Photran consiste em três passos: fazer uma pré-validação da re-

fatoração; fazer a manipulação da AST (introduzindo as mudanças no código-fonte); e fazer uma pós-validação da refatoração, para garantir a integridade da AST. O Photran possibilita atuar sobre o código Fortran por meio da manipulação de ASTs e da utilização da base de informações existentes nos VPGs. Existem métodos que possibilitam navegar sobre ASTs, recuperar informações acerca de seus nós, e ainda executar operações de atualização sobre elas (remoção, adição ou substituição de nós).

Photran oferece suporte à programação em linguagem Fortran 77, 90, 95, 2003 e 2008. Fortran foi a primeira linguagem de programação de alto nível, tornando-se uma linguagem de grande aceitação em computação científica (DE, 2004; ADAMS et al., 2008), sendo utilizada em áreas como previsão de tempo, análise de elementos finitos, dinâmica dos fluidos computacional, física computacional e química computacional. Mesmo tendo passado mais de cinquenta anos desde sua criação, a linguagem ainda é amplamente utilizada no meio científico.

Fortran oferece suporte para a utilização de bibliotecas para programação concorrente e distribuída como MPI (RASMUSSEN; SQUYRES, 2005) e OpenMP (HERMANNS, 2002), que viabilizam e facilitam a construção de programas paralelos. A linguagem Fortran tem sido utilizada para escrever diversos programas e rotinas em bibliotecas padronizadas, como BLAS (*Basic Linear Algebra Subprograms*), LAPACK (*Linear Algebra PACKage*), IMSL (*International Mathematics and Statistics Library*), BRAMS (*Brazilian Regional Atmospheric Modeling System*), NAG (*Numerical Algorithms Group*) e pode ser considerada a linguagem predominante em áreas de aplicação como matemática, física, engenharia e análises científicas (DE, 2004; KOFFMANN; FRIEDMAN, 2006; NYHOFF; LEESTMA, 1997).

Por ser uma ferramenta *open source*, possuir editor de código e *parser* para a linguagem Fortran, que por sua vez, possui suporte a programação concorrente com OpenMP e é amplamente utilizada na computação de alto desempenho, Photran possibilita e motiva seu uso no desenvolvimento deste trabalho.

Para implementar e integrar uma refatoração no Photran, é necessário estender o comportamento de algumas classes presentes em seu *framework*, como mostra a figura 3.4.

Essas classes são responsáveis por receber uma chamada do usuário e associar à ação de refatoração, criar uma interface gráfica para obter informações fornecidas pelo usuário e implementar a refatoração propriamente dita. No decorrer desta seção serão apresenta-

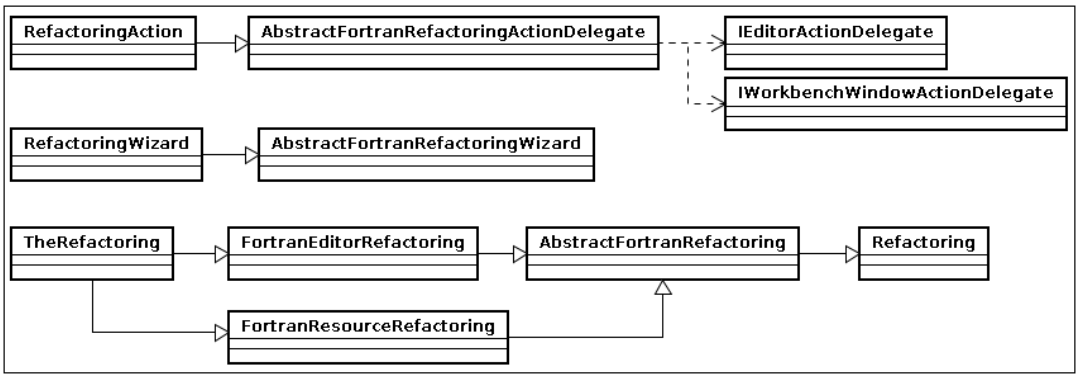

Figura 3.4: Diagrama de classes a serem estendidas (RISSETTI, 2011)

dos mais detalhes sobre as características e funcionalidades destas classes.

3.3.2 Definição do tipo de análise

A verificação de seções críticas identificará variáveis relacionadas ao trecho de código selecionado que podem ser envolvidas em uma condição de corrida durante a execução paralela do programa. Essa verificação foi dividida em dois níveis a serem escolhidos pelo programador:

- ***Exibir apenas as variáveis detectadas***
- ***Exibir todas as ocorrências das variáveis detectadas***

Para possibilitar ao programador a escolha do nível foi criada uma interface gráfica para a interação. Isso foi feito com a criação de uma classe de ação chamada *DetectPossibleCriticalSectionsAction* e de uma classe assistente da refatoração chamada *DetectPossibleCriticalSectionsWizard*.

A classe *DetectPossibleCriticalSectionsAction* refere-se à classe *RefactoringAction* apresentada na figura 3.4 e é responsável por receber a chamada do usuário e associar a refatoração a seu respectivo assistente. Essa classe deve estender a classe *AbstractFortranRefactoringActionDelegate* e deve implementar os métodos de duas interfaces: *IWorkbenchWindowActionDelegate*, permitindo que a chamada do usuário seja feita a partir do menu principal, e *IEditorActionDelegate*, que permite ao usuário fazer a chamada a partir de um menu de contexto no próprio editor de programas do Eclipse.

Já a classe *DetectPossibleCriticalSectionsWizard* refere-se à classe *RefactoringWizard* apresentada na figura 3.4. Essa classe estende *AbstractFortranRefactoringWizard* e tem o objetivo de fazer a construção gráfica da janela do assistente. Seu método construtor

recebe como parâmetro um objeto para a terceira classe necessária (*TheRefactoring*), que aplica a refatoração.

Ambas as classes, *DetectPossibleCriticalSectionsAction* e *DetectPossibleCriticalSectionsWizard*, são implementadas em um mesmo arquivo que deverá ser adicionado ao projeto org.eclipse.photran.ui.vpg. A figura 3.5 mostra a interface resultante dessa implementação.

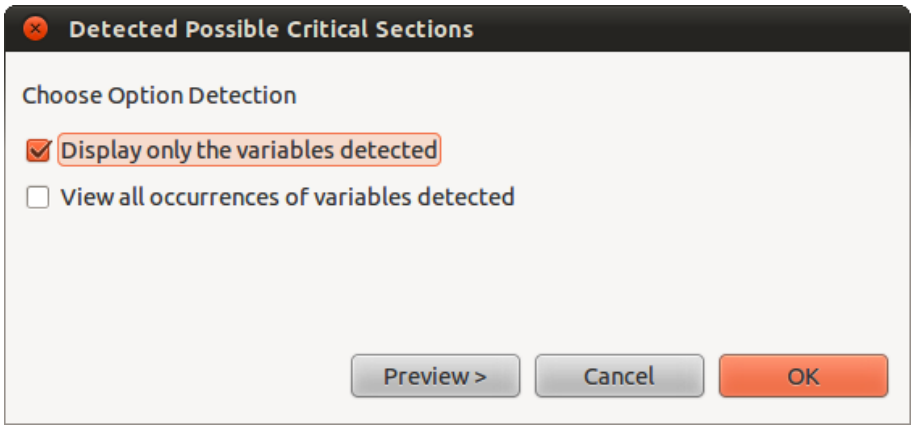

Figura 3.5: Definição do tipo de análise

A escolha entre um destes níveis interferirá na execução do algoritmo e também nos resultados apresentados. Tais interferências podem ser vistas na subseção 3.3.4.

3.3.3 Extensão da classe *TheRefactoring*

Com base no diagrama da figura 3.4, a classe *TheRefactoring* representa a refatoração a ser desenvolvida. Portanto, é necessário criar uma nova classe para executar o algoritmo proposto na estrutura de uma refatoração, denominada *DetectPossibleCriticalSections*. Essa é a classe principal, é nela que está desenvolvido o algoritmo.

3.3.3.1 Definição da categoria da refatoração

Para desenvolver uma refatoração no Photran primeiramente deve-se definir em qual categoria ela se enquadra. No Photran, as refatorações são divididas em duas categorias: refatoração baseada no editor, através da extensão da superclasse *FortranEditorRefactoring*, e refatoração no arquivo, através da extensão da superclasse *FortranResourceRefactoring*. A refatoração **baseada no editor** exige entradas do usuário, como a seleção de um trecho de código ao qual será aplicada a refatoração. Já na refatoração **baseada no arquivo**, o usuário pode selecionar vários arquivos de uma só vez, ou até mesmo pas-

tas inteiras ou projetos, e a refatoração será aplicada aos arquivos selecionados (CHEN; OVERBEY, 2010). Essas duas classes, por sua vez, estendem a classe *AbstractFortranRefactoring*, que, assim como as classes *FortranEditorRefactoring* e *FortranResourceRefactoring*, também é específica do Photran, e que, por sua vez, estende a classe *Refactoring* que faz parte do *framework* do Eclipse.

Como a proposta é analisar trechos de códigos que serão paralelizados, a refatoração deve ser baseada no editor tendo como entrada um trecho de código selecionado pelo programador. Portanto, a classe *DetectPossibleCriticalSections*, que implementa a refatoração, estende a superclasse *FortranEditorRefactoring*.

Após realizadas as implementações dessas classes, é necessário alterar o arquivo *manifest* (*plugin.xml*) para indicar ao Eclipse que existem novos pontos de extensão que precisam ser disponibilizados.

3.3.3.2 Codificação

Na classe *DetectPossibleCriticalSections*, que representa a refatoração, existem alguns métodos a serem codificados:

- ***getName*:** responsável por fornecer o título da refatoração, utilizado em janelas e em caixas de diálogo do Eclipse;
- ***doCheckInitialConditions*:** serve para fazer uma pré-validação da AST e dos critérios exigidos pela refatoração usada. Caso alguma condição não tenha sido atendida, o método pode gerar uma exceção indicando o problema e consequentemente abortando a execução da refatoração;
- ***doCreateChange*:** aplica a refatoração propriamente dita, fazendo as modificações na AST, caso a pré-validação tenha sido positiva. Ao término de tais modificações, esse método chama dois outros métodos: *addChangeFromModifiedAST()*, que tem o objetivo de adicionar as modificações feitas à AST utilizada pelo Photran e o método *releaseAllASTs()*, visando forçar o Photran a atualizar os controles visuais da AST assim como sua validação;
- ***doCheckFinalConditions*:** responsável por verificar as pós-condições, que confirmam ou não as alterações aplicadas sobre a AST. Esse método também pode lançar

uma exceção *PreconditionFailure*, indicando que alguma condição para a refatoração não foi satisfeita e que ela não será realizada.

Dos métodos citados, apenas *doCheckFinalConditions* não foi codificado, pois a refatoração não necessita de uma verificação final além das já realizadas pelo Photran nesse método. No método *getName* é necessário apenas definir uma *string* de retorno, *"Detect Possible Critical Sections"* foi definido como sendo o título da refatoração a ser retornado pelo método.

O método *doCheckInitialConditions* verifica se o programador selecionou um trecho de código contendo instruções a serem analisadas pelo algoritmo. Tal verificação é feita através da captura e da análise do texto selecionado pelo programador. Caso não tenha selecionado um trecho de código válido, uma mensagem de exceção é emitida ao programador. A mensagem pode ser vista na figura 3.6.

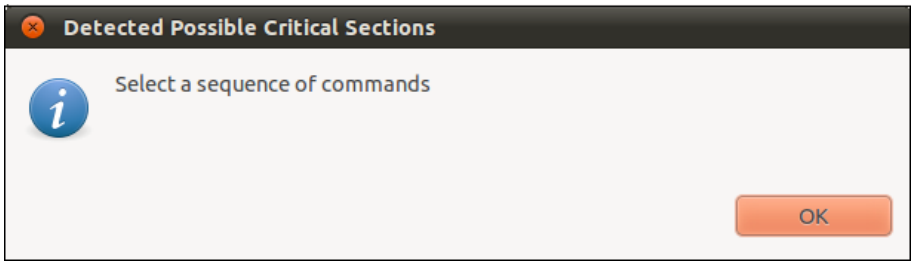

Figura 3.6: Mensagem de exceção

A mensagem da figura 3.6 é exibida nos seguintes casos:

- Não há um trecho de código selecionado ou o trecho selecionado está em branco. Um exemplo, é a seleção de linhas em branco apresentada na figura 3.7 a).

- O trecho de código selecionado não possui uma sequência de comandos para detecção, como por exemplo, a seleção de uma instrução de comentário Fortran apresentada na figura 3.7 b). Essa verificação é feita com a obtenção de uma sequência de comandos do trecho selecionado através de uma função disponibilizada pelo Photran. Caso não seja possível obter esses comandos, é gerada a mensagem de exceção.

Por fim, o método *doCreateChange* implementa a regra de detecção com base no algoritmo apresentado na seção 3.2. Apesar de ter sido desenvolvido como uma refatoração,

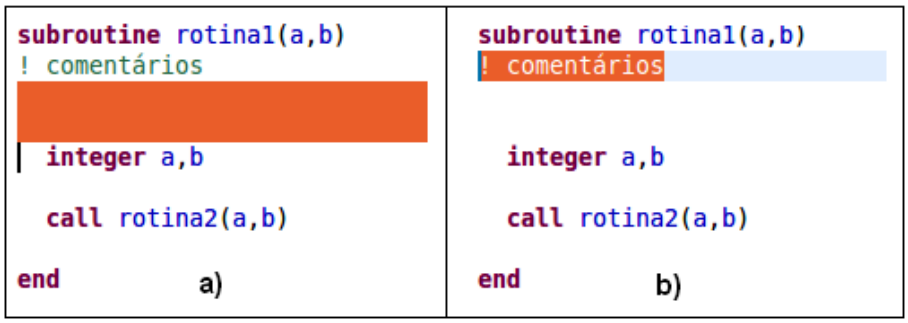

Figura 3.7: Casos de exemplo para a mensagem de exceção

isso não significa que o algoritmo seja uma refatoração propriamente dita. Nesse caso, apenas foi utilizada a estrutura provida pelo Photran para o desenvolvimento do detector.

Além da implementação do algoritmo, o método *doCreateChange* também faz a estruturação dos resultados da análise e inserções de sugestões de instruções OpenMP, através de alterações da AST. As seções 3.3.4 e 3.3.5 exibem os resultados e as sugestões de instruções OpenMP.

3.3.4 Exibição dos resultados

O resultado da análise é constituído de uma lista de variáveis e suas ocorrências no código. Essa lista é organizada na forma de tabela e seu número de ocorrências (linhas) está diretamente relacionado ao tipo de análise. Caso o programador selecione o tipo de análise *"Exibir apenas as variáveis detectadas"*, uma determinada variável terá uma única ocorrência dentro da tabela, ou seja, não haverá mais de uma linha referenciando a mesma variável. Já se for selecionada a opção *"Exibir todas as ocorrências das variáveis detectadas"*, serão referenciadas todas as ocorrências encontradas de uma determinada variável, portanto, poderá haver várias linhas contendo a mesma variável porém referenciando suas ocorrências dentro do código.

A exibição dos resultados é apresentada por uma estrutura criada separadamente do Photran. Essa estrutura foi desenvolvida como um novo *plugin* para o Eclipse, o qual foi chamado de ***Possible Critical Section*** e implementado em um novo pacote Java identificado como "*br.ufsm.inf.viewCriticalSection*".

A figura 3.8 mostra um exemplo de um trecho de código analisado, tendo como um dos resultados a tabela de variáveis detectadas na parte inferior da figura. A tabela resultante da detecção possui as seguintes colunas:

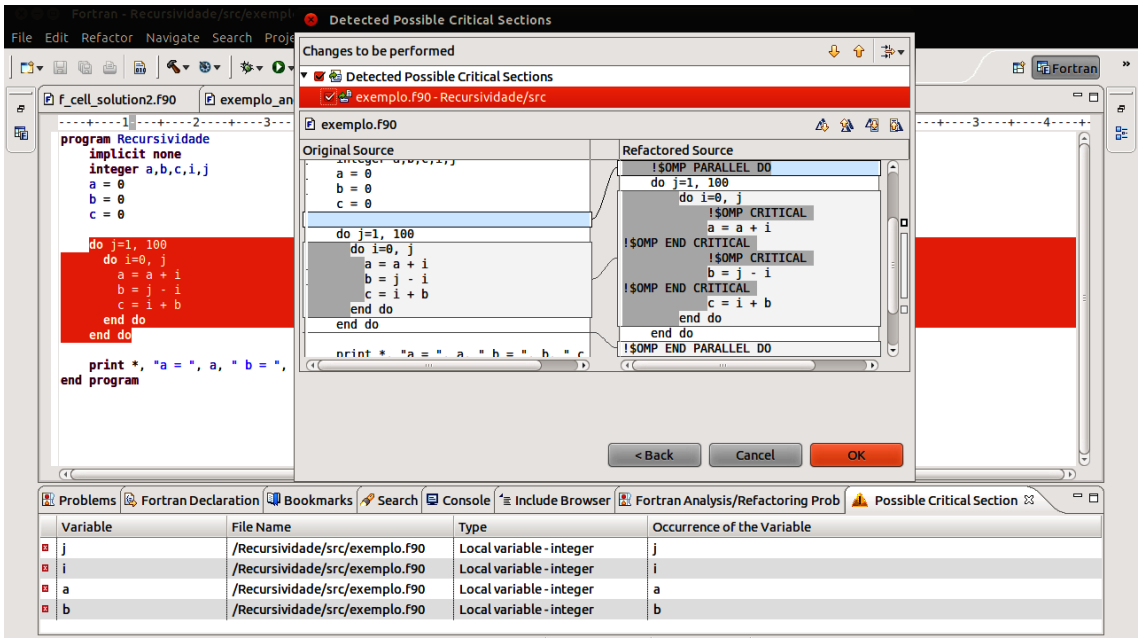

Figura 3.8: Detector

- ***Image***: Indentifica o tipo de resultado. A imagem na cor amarela indica uma possibilidade menor de acerto no resultado da detecção, sendo utilizada para variáveis como vetores e tipos derivados onde o detector ainda tem um nível baixo de precisão. Já a imagem na cor vermelha indica uma maior confiança no resultado da detecção.

- ***Variable***: Contém o nome da variável detectada presente no trecho de código selecionado ou em outra região do código, como por exemplo um procedimento chamado a partir do trecho, caso seja uma variável global.

- ***File Name***: Contém o nome do arquivo da ocorrência. Se o tipo de detecção for *Exibir apenas as variáveis detectadas* o nome do arquivo será referente à variável crítica, ou seja, o arquivo cujo o trecho de código foi selecionado. Já se o tipo de detecção for *Exibir todas as ocorrências das variáveis detectadas*, a coluna apresentará o nome do arquivo em que foi encontrada a ocorrência. Neste caso, o nome do arquivo pode ser diferente do arquivo onde foi selecionado o trecho de código.

- ***Type***: Identifica o tipo de variável detectada.

- ***Occurrence of the Variable***: Contém o nome da variável da ocorrência detectada ou a região do código, se for uma variável global encontrada fora do trecho selecionado.

- ***Marker***: Esse campo não é apresentado ao programador, apenas é mantido na tabela para guardar a posição do código fonte em que se encontra a ocorrência detectada. No momento em que o programador efetuar um duplo clique com o *mouse* sobre a linha da tabela, o editor do Eclipse posicionará e destacará a variável da ocorrência. A posição do marcador, assim como o nome do arquivo, vai depender do tipo de análise. Se for *"Exibir apenas as variáveis detectadas"*, guardará a posição da variável crítica. Já se o tipo de análise for "*"Exibir todas as ocorrências das variáveis detectadas"*, guardará a posição da ocorrência detectada.

Além de apresentar a tabela de resultados, a ferramenta implementada também faz algumas sugestões de instruções OpenMP. Estas sugestões foram implementadas principalmente para ilustrar a utilização do analisador como base para refatorações de código propriamente ditas.

3.3.5 Sugestões de instruções OpenMP

Os resultados do analisador abrem a possibilidade de inserir sugestões de instruções OpenMP. Mesmo não sendo o foco do trabalho, a adição de sugestões de instruções OpenMP foi implementada. Entretanto, ela é feita de forma simples, sem uma regra aprimorada para decidir qual é a melhor sugestão em cada caso.

As sugestões de instruções OpenMP são adicionadas na região do código onde se encontra o trecho selecionado para detecção. As instruções são comentários adicionados no código por meio de funcionalidades de manipulação da AST providas pelo Photran.

As primeiras instruções a serem adicionadas são: `!$omp parallel` em uma linha acima e `!$omp end parallel` em uma linha abaixo do trecho de código selecionado. Tais instruções são adicionadas apenas se o analisador gerar alguma ocorrência. Entretanto, se o trecho de código selecionado for uma laço de repetição "DO", as instruções adicionadas serão respectivamente `!$omp parallel do` e `!$omp end parallel do`

Caso a variável detectada seja o índice do laço de repetição, adiciona-se a variável em uma lista de variáveis *private*. Havendo elementos nessa lista ao terminar o processo de detecção, eles são adicionados na instrução inicial precedidos pela instrução *private*. Por exemplo, `!$omp parallel private(i,j)`.

Na situação em que o trecho de código selecionado for um laço repetição "DO", a

variável de controle do laço não será adicionada à lista de variáveis *private*. Isso deve-se ao fato de que o OpenMP, por padrão, trata essa variável como *private* na instrução `!$omp parallel do`.

Nos nós onde há a atribuição de valores em uma variável e esta foi indicada pelo detector, é adicionada a instrução OpenMP `!$omp critical` na linha acima e `!$omp end critical` na linha abaixo da instrução.

Nas chamadas de sub-rotinas que foram identificadas variáveis com risco de envolvimento em uma condição de corrida, são adicionados comentários de aviso acima da chamada. Esse comentário contém uma mensagem de aviso e a lista de variáveis em risco, como por exemplo:

```
!*** Warning ***
!Critical variables identified in the call: a, b, c
call getArea(a,b,c)
```

Algumas das instruções apresentadas podem ser vistas na figura 3.8. Embora as sugestões sejam feitas de forma simples, o mecanismo de inserção destas instruções está implementado. Todavia, carece ainda de regras mais bem elaboradas e precisas. A melhoria na qualidade das sugestões deve ser um dos próximos passos na continuidade da pesquisa e implementação do analisador, passando por um estudo aprofundado em relação a instruções OpenMP e, consequentemente, a criação de um algoritmo que proporcione uma melhor precisão na decisão das instruções a serem sugeridas.

O mecanismo de sugestões de instruções OpenMP completa o processo de desenvolvimento do analisador. O capítulo seguinte apresentará os testes realizados com a ferramenta.

4 AVALIAÇÃO

Este capítulo apresenta uma avaliação empírica do analisador proposto neste trabalho, mostrando as situações nas quais ele é capaz de detectar possíveis problemas de condição de corrida e também eventuais falsos positivos e falsos negativos. Além disso, são avaliadas as sugestões de instruções OpenMP feitas pelo analisador, verificando se elas são adequadas às situações apresentadas.

O funcionamento do analisador foi avaliado em duas etapas. Na primeira etapa, foram realizados testes básicos, com curtos trechos de código e pequenas aplicações disponibilizadas para testes com OpenMP. Na segunda etapa, realizou-se um estudo de caso aplicando o analisador a um programa Fortran de grande porte. O caso escolhido foi o modelo OLAM (WALKO; AVISSAR, 2008), uma aplicação de computação científica utilizada em produção e em pesquisa.

4.1 Testes básicos

Os testes básicos foram divididos em duas partes: a primeira com pequenos exemplos de código analisados e a segunda com pequenas aplicações de testes que já contêm trechos paralelizados com OpenMP.

4.1.1 Exemplos

Os exemplos apresentados nesta subseção foram criados única e exclusivamente para testar o comportamento do analisador. Os trechos de código selecionados pelo usuário programador estão destacados em vermelho nas figuras.

O primeiro exemplo (figura 4.1), é um caso simples para validar o funcionamento básico do algoritmo, que consiste na verificação de escrita e leitura sobre um determinado dado. Neste exemplo, a variável `x` foi identificada pelo detector como uma variável com

um potencial risco de corrida, por possuir uma operação de escrita e também de leitura sobre o dado que representa.

```
subroutine example1(x)
    x = x + 1
end subroutine
```

Figura 4.1: Exemplo 1: variável lida e alterada

O segundo exemplo apresenta um trecho de código mais complexo que o primeiro. Nesse trecho, é necessário examinar o código da sub-rotina chamada dentro do trecho de código selecionado. O exemplo pode ser visto na figura 4.2.

```
subroutine example2()
  integer x,y,w
  x = 10
  y = 20
  call example2_ab(x,y)
  w = x + y
end subroutine

subroutine example2_ab(a,b)
  a = a + 2
  b = a * 5
end subroutine
```

Figura 4.2: Exemplo 2: chamada de sub-rotina

No segundo exemplo, as variáveis `x` e `y` foram detectadas devido ao fato de que seus respectivos valores estão sendo alterados na sub-rotina `exemplo2_ab`, através das variáveis `a` e `b`. Além disso, esses valores estão sendo lidos na instrução `w = x + y` do trecho selecionado e na variável `a` dentro da sub-rotina.

A verificação de sub-rotinas é feita usando os conceitos de recursividade. Se, durante a verificação de um trecho de código existir uma chamada para uma sub-rotina, o método de verificação do trecho é novamente chamado, passando o trecho de código da sub-rotina para verificação. Após a verificação da sub-rotina, o trecho anterior continua sendo verificado.

Até então, foram apresentadas situações nas quais são detectadas variáveis que podem realmente apresentar problemas durante a execução paralela do trecho analisado. Entretanto, o analisador pode apresentar alguns resultados que não representam perigo, os quais são chamados falsos positivos. A figura 4.3 apresenta uma situação na qual isto ocorre.

```
subroutine example()
  integer x,y,w
  x = 10
  y = 20
  call example2_ab(x,y)
  w = x + y
end subroutine

subroutine example2_ab(a,b)
  a = 2
  b = 5
end subroutine
```

Figura 4.3: Exemplo 3: falso positivo

O exemplo da figura 4.3 é muito semelhante ao segundo. Alterou-se apenas a sub-rotina `example2_ab`, onde as variáveis `a` e `b` passaram a receber valores constantes. Nesse caso, o resultado da análise foi o mesmo do segundo exemplo, ou seja, identificou-se as variáveis `x` e `y` como potenciais problemas em uma execução paralela. Mas, como `x` e `y` estão recebendo valores constantes, o resultado da execução paralela não se alterará, independentemente da ordem e do tempo em que as *threads* executarem. Sendo assim, os resultados apresentados pelo analisador representam falsos positivos.

O analisador busca também por variáveis globais que estão sendo escritas e lidas. Se essas operações, escrita e leitura, sobre uma determinada variável global acontecerem dentro do trecho selecionado, a situação será a mesma do primeiro exemplo (figura 4.1). Porém, essas operações podem acontecer em sub-rotinas e a variável global envolvida pode nem estar presente no trecho de código selecionado, mas com risco de ser envolvida em uma condição de corrida alterando o resultado da execução.

A figura 4.4 mostra uma situação na qual uma variável global pode ser envolvida em uma condição de corrida. Esse exemplo é semelhante ao segundo exemplo, mas nesse caso as variáveis `x` e `y` não são passadas como parâmetro para a sub-rotina e foram detectadas por serem variáveis globais, com operação de escrita na sub-rotina e leitura no trecho selecionado.

Paralelizações de trechos de códigos são muito utilizadas em laços de repetição, dividindo o trabalho entre as *threads*, ficando cada *thread* responsável pela execução de uma parte dos índices do laço. A figura 4.5 mostra a seleção de um trecho de código para análise, constituído de laços de repetição.

```
module module_example3
  implicit none
  integer x,y
end module

subroutine example3()
  use module_example3
  integer w
  call example3_global()
  w = x + y
end subroutine

subroutine example3_global()
  use module_example3
  x = x + 2
  y = x * 5
end subroutine
```

Figura 4.4: Exemplo 4: variável global

```
subroutine example4
  implicit none
  integer a,i,j
  a = 0
  do j=1, 100
      do i=0, j
          a(j) = a(j) + i
      end do
  end do
end subroutine
```

Figura 4.5: Exemplo 5: laço de repetição

Nesse caso, o analisador identificou as variáveis `j`, `i` e `a()`. Com relação aos índices `j` e `i`, a detecção está correta pois há leitura e escrita sobre essas variáveis, embora na paralelização desse trecho com OpenMP, a instrução `parallel do` colocada antes do primeiro laço já sirva para privatizar o índice `j` por padrão. Quanto à variável `a()`, apesar de possuir operações de escrita e leitura, essas operações ocorrem em um mesmo índice do vetor, que é controlado por `j`, não representando portanto um possível problema de condição de corrida. Isso constitui um falso positivo apresentado pelo detector.

Nessa subseção foram apresentados alguns exemplos que validam o detector e mostram algumas situações em que podem ser gerados falsos positivos. Nesses exemplos não foram detectados falsos negativos, ou seja, situações que representam risco de condição de corrida que o detector não foi capaz de detectar. A próxima subseção mostrará a

validação da ferramenta com o uso de pequenas aplicações de testes com OpenMP.

4.1.2 Aplicações OpenMP

Os testes realizados com as aplicações já paralelizadas com OpenMP focaram-se nos trechos já paralelizados dos programas *Pi*, *Cellular Automata*, *Jacobi* e *Molecular Dynamic*. Esses programas fazem parte do repositório OmpSCR - *OpenMP Source Code Repository* (DORTA; RODRÍGUEZ; SANDE, 2009), um repositório de códigos-fonte livres em OpenMP, úteis para testes.

O programa *Pi* trabalha com o cálculo do número Pi. Nele foi selecionado um trecho simples de código (figura 4.6).

```
!$OMP PARALLEL PRIVATE(local,i), SHARED(w,N), &
!$OMP REDUCTION(+:pi)
!$OMP DO
do i=1,N
  local = (i + 0.5) * w
  pi = pi + 4.0 / (1.0 + local * local)
end do
!$OMP END DO
!$OMP END PARALLEL
```

Figura 4.6: Exemplo: cálculo de Pi

Ao analisar o trecho de código da figura 4.6, o detector retornou corretamente como resultado três variáveis com risco de condição de corrida, considerando a execução paralela do trecho. As variáveis e as sugestões de proteção podem ser vistas na figura 4.7.

Em relação às sugestões OpenMP, a proteção `critical` sugerida pelo detector para variável `local` não foi adequada nessa situação, pois não evita o problema de condição de corrida. Essa variável deveria ter um outro tipo de proteção, como por exemplo `private`, assim como nas diretivas já existentes para o trecho antes da análise do detector (figura 4.6). As demais sugestões estão de acordo, embora, para a variável `pi`, a melhor opção possa ser a redução, mas a sugestão dessa instrução envolveria análises que não foram aprofundadas neste trabalho. As instruções de início e fim do trecho paralelizado foram inseridas corretamente e a proteção da variável `i` é feita por padrão na paralelização do laço de repetição, utilizando a instrução `!$OMP PARALLEL DO`.

O próximo teste é com o programa *Cellular Automata*, que trabalha com autômatos celulares. Nesse teste, foi selecionado o trecho de código apresentado pela figura 4.8 para

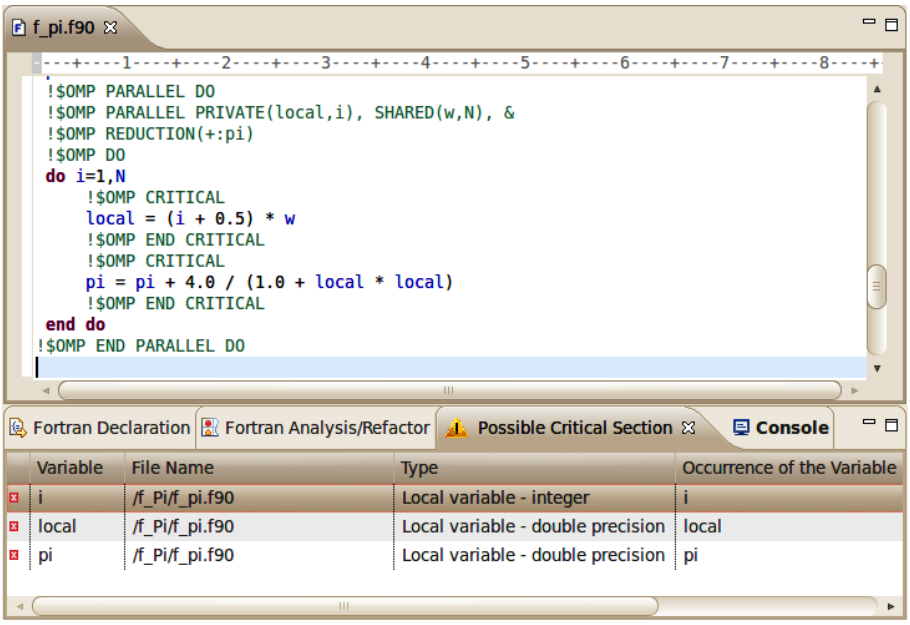

Figura 4.7: Resultado: cálculo de Pi

ser analisado pelo detector. O resultado da análise é apresentado na figura 4.9.

```
!$OMP PARALLEL default(none) shared(numiter,nthreads,size_y,size_x,M,
                              oldM) private(iter,thread,limitL,limitR,i,j)
thread = OMP_GET_THREAD_NUM()
limitL = thread*(size_y/nthreads)+1
limitR = (thread+1)*(size_y/nthreads)
oldM(1:size_x,limitL:limitR) = M(1:size_x,limitL:limitR)

DO i=1,size_x
   DO j=limitL,limitR
      M(i,j) = (                          &
      oldM(i-1,j) +                  &
      oldM(i+1,j) +                  &
      oldM(i,j-1) +                  &
      oldM(i,j+1) ) / 4.0
   ENDDO
ENDDO
!$OMP END PARALLEL
```

Figura 4.8: Exemplo: Cellular Automata

Como pode ser observado na figura 4.9, o analisador retornou como resultado sete variáveis que podem vir a ser envolvidas em uma condição de corrida e, por isso, devem ser analisadas com cuidado pelo programador, para definir as proteções necessárias para as mesmas.

Das variáveis detectadas, duas possuem a imagem amarela na tabela de resultados por se tratarem de matrizes. Nestes casos o detector tem uma precisão baixa, podendo

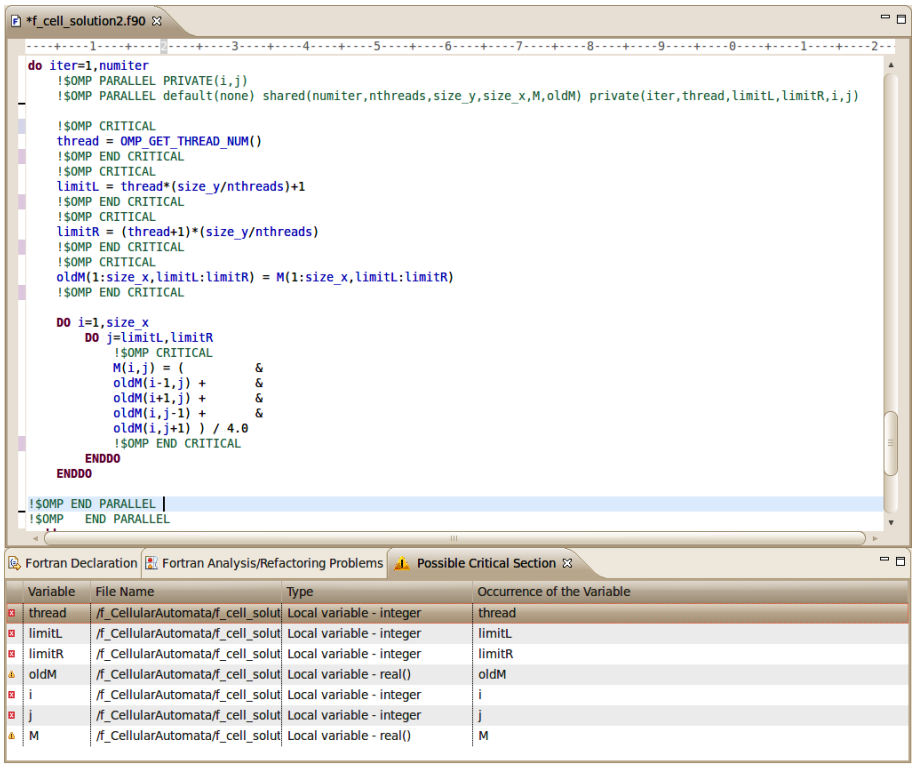

Figura 4.9: Resultado: Cellular Automata

a variável não apresentar riscos reais de condição de corrida. As demais variáveis estão identificadas com a imagem vermelha, onde há uma maior probabilidade das variáveis apresentarem problemas.

Analisando o código e utilizando como base as instruções OpenMP já contidas no mesmo, as variáveis em vermelho estão todas protegidas como privadas e as amarelas estão compartilhadas. Sendo assim, dos resultados apresentados, somente as variáveis em amarelo representam falsos positivos.

Quanto à sugestão de instruções OpenMP, as instruções de início e fim do trecho paralelizado foram inseridas corretamente pela ferramenta, entretanto, os trechos sugeridos como *critical* foram inadequados para esta situação. As variáveis envolvidas nesta situação poderiam ser sugeridas como privadas ao invés de definir um trecho crítico, como feito para as variáveis `i` e `j`.

O programa *Jacobi* trabalha com a equação de Helmholtz, usando o método iterativo de *Jacobi*. Neste programa foram selecionados quatro trechos de códigos para serem analisados. Nestes trechos, foram detectadas corretamente todas as variáveis em risco. Para todos os casos, a tabela de resultados apresentou somente variáveis identificadas com a imagem vermelha. Entretanto, a sugestão da instrução `!$omp critical` não está ade-

quada na maioria dos casos, assim como ocorreu nos testes apresentados anteriormente.

O programa *Molecular Dynamic* completa a lista de programas simples testados. Este programa implementa uma simples simulação de dinâmica molecular. Nele foram selecionados dois trechos de código, sendo que em um deles, apresentado pela figura 4.10, há chamadas para sub-rotinas.

```
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i,j,k,rij,d) &
!$OMP REDUCTION(+ : pot, kin)
do i=1,np
   f(1:nd,i) = 0.0
   do j=1,np
      if (i > j) then
         call dist(nd,box,pos(1,i),pos(1,j),rij,d)
         pot = pot + 0.5*v(d)
         do k=1,nd
            f(k,i) = f(k,i) - rij(k)*dv(d)/d
         enddo
      endif
   enddo
   kin = kin + dotr8(nd,vel(1,i),vel(1,i))
enddo
!$omp end parallel do
```

Figura 4.10: Exemplo: Molecular Dynamic

A figura 4.11 mostra o resultado da verificação do trecho. Em uma das chamadas foram detectadas corretamente duas variáveis: `rij` e `d`. A primeira variável é um vetor e está identificada na tabela de resultados pela imagem amarela, já a segunda é uma variável de ponto flutuante identificada na tabela de resultados pela imagem vermelha. Nesse teste foram identificadas corretamente oito variáveis, sendo duas delas identificadas pela imagem amarela por se tratarem de um vetor e uma matriz.

Nos testes realizados com os programas *Pi*, *Cellular Automata*, *Jacobi* e *Molecular Dynamic* foram analisados oito trechos de código, onde o detector identificou trinta e nove variáveis com possibilidade de apresentar condição de corrida. A tabela 4.1 apresenta com mais detalhes os dados estatísticos desses testes.

Das trinta e nove variáveis detectadas, seis são falsos positivos, o que representa 15% das variáveis. Todas as variáveis na situação de falso positivo foram identificadas com a imagem amarela na tabela de resultados, indicando que nessas situações o analisador tem uma precisão baixa de acerto. Vale ressaltar também, que nesses testes não houve a ocorrência de falsos negativos, ou seja, o analisador não deixou de detectar variáveis com

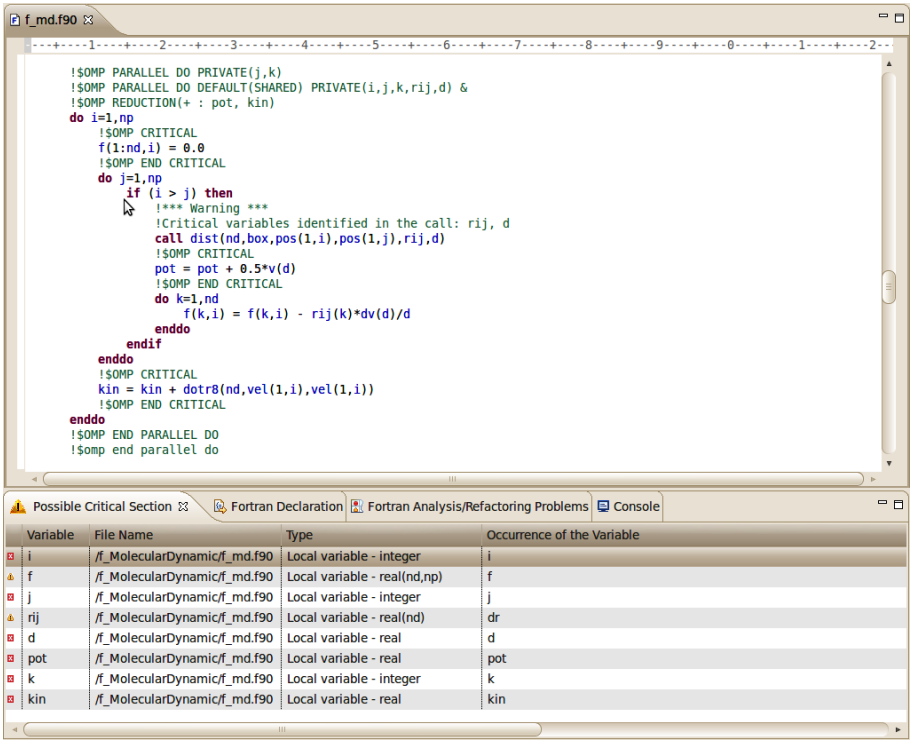

Figura 4.11: Resultado: Molecular Dynamic

Tabela 4.1: Variáveis detectadas

	Nº de Variáveis	Percentual
Acertos	33	85%
Falsos Positivos	6	15%
Falsos Negativos	0	0%
Total	39	100%

risco de condição de corrida. O restante das variáveis representam acertos do detector, somando trinta e três variáveis que representam 85% das variáveis detectadas.

Os testes apresentados até então tiveram resultados positivos, entretanto, os códigos utilizados são de baixa complexidade. O próximo estudo de caso, realizado sobre o programa OLAM, representa uma avaliação do analisador em um código de grande porte.

4.2 Estudo de Caso: OLAM

O OLAM (*Ocean-Land Atmosphere Model*) é um modelo de simulação numérica para a climatologia, baseado no Regional Atmospheric Modeling System (RAMS) e desenvolvido pela Duke University. Sua principal característica é a capacidade de representar fenômenos meteorológicos de escala global e, com o acoplamento de grades refinadas, representar de forma mais precisa os fenômenos de escala local e estimar o clima regional

(WALKO; AVISSAR, 2008)

OLAM foi desenvolvido em Fortran 90 e constitui-se de diversas sub-rotinas, módulos e os mais distintos tipos de estruturas de dados e de instruções Fortran. Entre seus vários arquivos, existem cento e oitenta (180) arquivos com código Fortran, que juntos possuem um total de duzentas e dezoito mil quinhentas e cinquenta e oito (218.558) linhas. Trata-se, portanto, de um caso grande e complexo, representativo do tipo de aplicação que pode se beneficiar do analisador desenvolvido.

Assim como os exemplos da seção 4.1.2, o OLAM também possui trechos de códigos já paralelizados com instruções OpenMP. É nesses trechos de código que os testes foram concentrados, para permitir uma comparação entre a paralelização manual e a paralelização com auxílio do analisador. Num primeiro momento, foram identificados tais trechos de código e, em seguida, foi feita a análise dos trechos com o detector.

A figura 4.12 mostra um dos trechos analisados. Nesse exemplo, há dois laços de repetição nas quais as variáveis que representam seus índices devem ser protegidas, porém o primeiro laço representa o trecho selecionado. Com isso, basta inserir a instrução `!$omp parallel do` e o índice desse laço será protegido por padrão. Outras variáveis que podem apresentar problemas durante a execução paralela deste trecho são `iw` e `ka` que estão sendo escritas e lidas.

```
!$omp parallel do private(iw,ka,k)
do j = 1,jtab_w(17)%jend(mrl); iw = jtab_w(17)%iw(j)
!------------------------------------------------
   call qsub('W',iw)

   ka = lpw(iw)

   do k = ka,mza-1

      alpha_press(k,iw) = pc1                                &
         * (((1. - sh_w(k,iw)) * rdry + sh_v(k,iw) * rvap)  &
         * theta(k,iw) / thil(k,iw)) ** cpocv

   enddo

enddo
!$omp end parallel do
```

Figura 4.12: OLAM – exemplo 1

Como pode ser visto na figura 4.13, o analisador identificou corretamente todas as variáveis com risco de condição de corrida desse trecho. Porém, a sugestão de instruções

OpenMP não teve a mesma precisão da detecção, pois a instrução `!$OMP CRITICAL` não foi adequada para as variáveis `iw` e `ka`. Uma solução seria que tais variáveis fossem privatizadas, assim como a variável `K`.

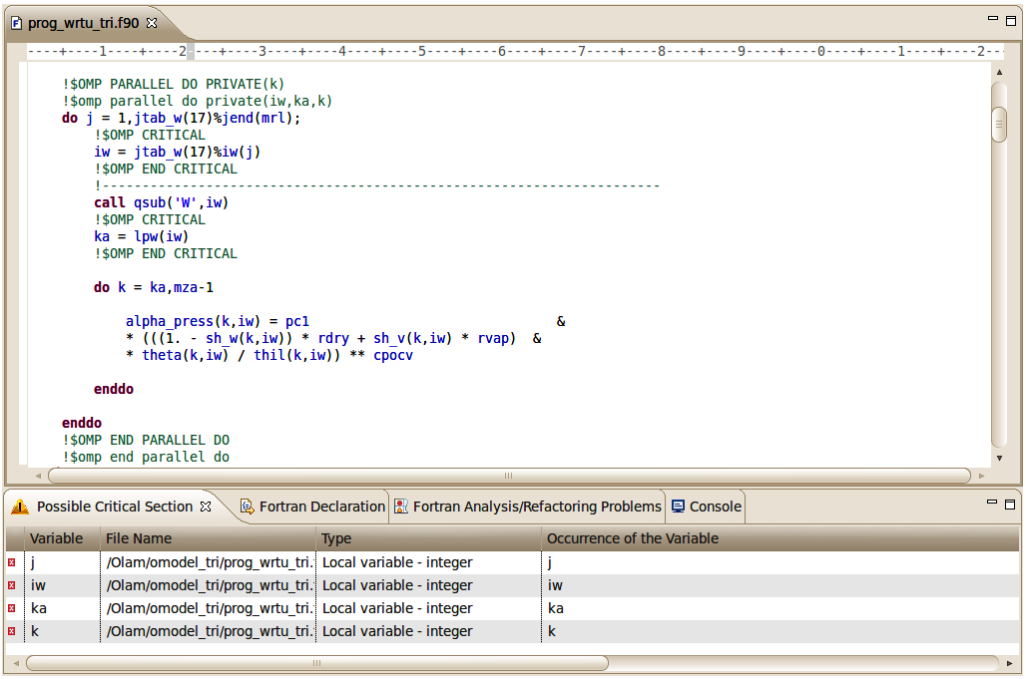

Figura 4.13: OLAM – resultado do exemplo 1

O exemplo acima representa um pequeno trecho de código, apesar de haver uma chamada de sub-rotina onde seu código deve ser analisado, e com poucas variáveis com risco de corrida. Mas no OLAM também foram analisados trechos maiores de código, onde há uma quantidade significativa de variáveis em risco. Um exemplo desses trechos pode ser visto na figura 4.14.

Nesse segundo exemplo, há 26 variáveis privatizadas com a instrução `private` no início do trecho, sem contar o índice `j` do primeiro laço que está protegido por padrão com o uso a instrução `!$omp parallel do`. São, portanto, vinte e sete (27) variáveis em condição de risco.

Nesse trecho, analisador identificou as 29 variáveis, sendo 27 acertos e dois falsos positivos. Isto representa uma aproximadamente 93% de acerto. Os dois falsos positivos gerados estão indicados pela imagem amarela na tabela de resultados. Porém, mais uma vez, a sugestão da instrução `!$omp critical` não foi adequada em algumas situações. As demais instruções foram inseridas corretamente.

Os testes com o programa OLAM totalizaram 100 trechos de código, que por sua vez,

```
!$omp parallel do private(iv,iw1,iw2,kb,k,dto2,dxps1,dyps1,dxps2, &
!$omp                     dyps2,cosv1,sinv1,cosv2,sinv2,wnx_v,wny_v, &
!$omp                     wnz_v,vxeface,vyeface,vzeface,uface,wface)
do j = 1,jtab_v(15)%jend(mrl); iv = jtab_v(15)%iv(j)
  iw1 = itab_v(iv)%iw(1); iw2 = itab_v(iv)%iw(2)
  !-----------------------------------------------------------------
  call qsub('V',iv)
  kb = lpv(iv)
  dto2 = .5 * dtm(itab_v(iv)%mrlv)
  dxps1 = itab_v(iv)%dxps(1)
  dyps1 = itab_v(iv)%dyps(1)
  dxps2 = itab_v(iv)%dxps(2)
  dyps2 = itab_v(iv)%dyps(2)
  cosv1 = itab_v(iv)%cosv(1)
  sinv1 = itab_v(iv)%sinv(1)
  cosv2 = itab_v(iv)%cosv(2)
  sinv2 = itab_v(iv)%sinv(2)
  wnx_v = xev(iv) * eradi
  wny_v = yev(iv) * eradi
  wnz_v = zev(iv) * eradi
  ! Vertical loop over T/V levels
  do k = kb,mza-1
    ! Average 3 earth velocity components from T points to V face
    vxeface = .5 * (vxe(k,iw1) + vxe(k,iw2))
    vyeface = .5 * (vye(k,iw1) + vye(k,iw2))
    vzeface = .5 * (vze(k,iw1) + vze(k,iw2))
   !Project earth velocity components at V face onto U and W directions
    uface = unx(iv) * vxeface + uny(iv) * vyeface + unz(iv) * vzeface
    wface = wnx_v   * vxeface + wny_v   * vyeface + wnz_v   * vzeface
    ! Compute displacement components for V face relative to T point
    if (vs(k,iv) > 0.) then
      iwdepv(k,iv) = iw1
      dxps_v(k,iv) = -dto2 * (vs(k,iv) * cosv1 - uface * sinv1) + dxps1
      dyps_v(k,iv) = -dto2 * (vs(k,iv) * sinv1 + uface * cosv1) + dyps1
      dzps_v(k,iv) = -dto2 * wface
    else
      iwdepv(k,iv) = iw2
      dxps_v(k,iv) = -dto2 * (vs(k,iv) * cosv2 - uface * sinv2) + dxps2
      dyps_v(k,iv) = -dto2 * (vs(k,iv) * sinv2 + uface * cosv2) + dyps2
      dzps_v(k,iv) = -dto2 * wface
      endif
    enddo
  enddo
```

Figura 4.14: OLAM – exemplo 2

já estavam paralelizados com OpenMP. A tabela 4.2 apresenta os dados estatísticos dos testes realizados com o OLAM. Foram detectadas 688 variáveis com risco de condição de corrida. Dessas variáveis, 161 são falsos positivos, o que representa 23% das variáveis detectadas. As demais variáveis, 527, são os acertos representando 77% das variáveis. Assim como nos testes anteriores, não foram identificados falsos negativos, ou seja, apa-

rentemente o detector não deixou de apontar nenhuma variável em risco.

Tabela 4.2: OLAM – variáveis detectadas

	Nº de Variáveis	Percentual
Acertos	527	77%
Falsos Positivos	161	23%
Falsos Negativos	0	0%
Total	688	100%

Dos 161 falsos negativos gerados, apenas cinco ocorrências foram identificadas com a imagem vermelha na tabela de resultados do analisador, sendo todas elas referentes à mesma variável. A grande maioria foi identificada com a imagem amarela, representando uma menor precisão do analisador.

Com relação as sugestões de instruções OpenMP, a identificação de início e fim do trecho paralelizado através da instrução `!$omp parallel do`, por se tratarem de laços de repetição, foi inserida corretamente. A privatização de variáveis utilizando `private` e a listagem das variáveis em risco contidas em uma sub-rotina através de um comentário também foram feitas corretamente. Entretanto, como nos testes anteriores, a sugestão da instrução `!$omp critical` não foi adequada em algumas situações.

4.3 Considerações sobre a avaliação

De modo geral, os testes apresentaram resultados positivos. Obteve-se 85% de acertos nos primeiros testes, utilizando programas simples com trechos em OpenMP, e 77% de acerto no estudo de caso com o programa OLAM. Os falsos positivos apresentados pelo analisador representam, na sua grande maioria, estruturas de dados nas quais o analisador ainda não possui uma boa precisão de acerto. Assim, era esperado que fossem gerados falsos positivos para esses tipos de dados durante os testes. Um aspecto relevante da avaliação foi que não foram identificados falsos negativos. Assim, supõe-se que o analisador não tenha deixado de detectar nenhuma variável com risco de corrida.

As sugestões de instruções OpenMP, com exceção da instrução `!$omp critical`, foram inseridas corretamente, com grande possibilidade de serem mantidas pelo o programador para a paralelização do trecho. A implementação da instrução `!$omp critical` necessita ser revista, pois sua utilização em muitos casos não foi adequada.

Através dos testes e do estudo de caso realizados, foi possível mostrar e avaliar o

uso do analisador. Apesar de possuir limitações, o analisador apresentou bons resultados, detectando, até onde foi possível verificar, todas as variáveis em risco de se envolverem em uma condição de corrida na execução paralela dos trechos analisados.

5 CONCLUSÃO

Nesta dissertação, foi explorada a refatoração de código sequencial em código paralelo OpenMP – tarefa essa que exige tempo, conhecimento e muita atenção do programador. O objetivo do trabalho é auxiliar na refatoração, dando mais agilidade e segurança ao processo. Para isso, foi proposto e implementado um algoritmo para análise automática de acessos concorrentes a dados, com a finalidade de identificar variáveis que possam vir a ter problemas de corrida.

O algoritmo proposto pode ser utilizado em programas Fortran e, possivelmente, em outros programas sequencias desenvolvidos com linguagens procedurais similares ao Fortran que se deseja paralelizar para execução em arquiteturas computacionais de memória compartilhada. Ele necessita do auxílio de um analisador sintático para obter e navegar na árvore sintática do código-fonte. Seu funcionamento consiste na entrada de um trecho de código do programa selecionado pelo programador. Com base nesse trecho, são analisados os acessos de leitura e escrita aos dados, tendo como saída as variáveis e a sua localização no código que caracteriza a possibilidade de uma condição de corrida em uma execução paralela do trecho.

Para implementação da ferramenta de análise foi utilizado o Photran, um ambiente integrado de desenvolvimento (IDE) para código Fortran baseado na plataforma Eclipse, que disponibiliza um *framework* para o rápido desenvolvimento de refatorações. O Photran possui um analisador sintático que possibilitou a automatização do algoritmo proposto através da implementação de uma refatoração, denominada *"Detect Possible Critical Sections"*.

A ferramenta desenvolvida utiliza o algoritmo para a análise do código, exibindo as variáveis que podem ter problema de corrida e informações adicionais relativas à localização da variável. Além disso, a ferramenta faz sugestões de diretivas OpenMP para o

programa paralelo, baseadas no resultado da análise do algoritmo. Sua utilização auxilia o programador no processo de refatoração de código sequencial em código paralelo OpenMP reduzindo os riscos de erros, além de reduzir também o tempo e o trabalho de refatoração.

A ferramenta foi avaliada em pequenos trechos de código, pequenas aplicações contendo instruções OpenMP e no programa OLAM, todos contendo código-fonte Fortran. Os testes utilizando trechos de código e pequenas aplicações OpenMP foram realizados para mostrar o que a ferramenta é capaz de fazer e também alguns problemas já previstos na implementação, sendo esses problemas basicamente relacionados a falsos positivos que possam vir a ser apresentados. Já no teste com o programa OLAM, buscou-se fazer uma avaliação mais consistente da ferramenta, que obteve bons resultados. Nos testes realizados, não houve situações nas quais a ferramenta tenha deixando de identificar variáveis que realmente apresentam risco de corrida e obteve-se um percentual aceitável de falsos positivos, sendo de 23% nos testes com o OLAM e 15% nos testes com pequenos programas OpenMP. Os falsos positivos apresentados estão relacionados, na sua grande maioria, com estruturas de dados nas quais a ferramenta conhecidamente possui uma baixa precisão, sendo essas ocorrências identificadas de forma diferente das situações nas quais há maior precisão na apresentação dos resultados.

Embora a sugestão de diretivas OpenMP não fosse o objetivo principal do trabalho, os resultados da análise viabilizam esse tipo de funcionalidade. As sugestões foram implementadas de uma forma simples, sem um algoritmo de decisão apropriado para a escolha da melhor opção de sugestão. Devido a isso, durante os testes houveram sugestões não adequadas às situações em que foram propostas, principalmente com relação à instrução `critical`. Também houve situações nas quais seria mais adequada a utilização de outras diretivas OpenMP que não foram incluídas na implementação, como por exemplo a diretiva `reduction`. Como fato positivo, está a viabilidade da implementação de sugestões OpenMP baseadas no resultado da análise para auxiliar na refatoração de programas seriais para paralelos.

Como trabalhos futuros, pretende-se propor e integrar à ferramenta um algoritmo adequado para sugestão de diretivas OpenMP baseado no resultado da análise. Além disso, serão estudas alternativas para trabalhar com a análise de acesso em variáveis com estrutura de dados organizadas em vetores, matrizes e tipos derivados, a fim de reduzir o

número de ocorrências de falsos positivos.

REFERÊNCIAS

ADAMS, J. C.; BRAINERD, W. S.; HENDRICKSON, R. A.; MAINE, R. E.; MARTIN, J. T.; SMITH, B. T. **The Fortran 2003 Handbook**: the complete syntax, features and procedures. New York: Springer Publishing Company, Inc., 2008.

ANDRONIKOS, T.; CIORBA, F. M.; THEODOROPOULOS, P.; KAMENOPOULOS, D.; PAPAKONSTANTINOU, G. Cronus: a platform for parallel code generation based on computational geometry methods. **Journal of Systems and Software**, [S.l.], v.81, n.8, p.1389–1405, 2008.

ARB and. **OpenMP Application Program Interface**. Version 3.0.ed. [S.l.]: OpenMP ARB, 2008. Disponível em: http://www.openmp.org/mp-documents/spec30.pdf. Acesso em: Novembro de 2011.

ARNOLD, R. S. An Introduction to Software Restructuring. In: ARNOLD, R. S. (Ed.). **Tutorial on Software Restructuring**. [S.l.]: IEEE Press, 1986.

BANERJEE, P.; CHANDY, J.; GUPTA, M.; HODGES, E. I.; HOLM, J.; LAIN, A.; PALERMO, D.; RAMASWAMY, S.; SU, E. The Paradigm compiler for distributed-memory multicomputers. **Computer**, [S.l.], v.28, n.10, p.37–47, oct 1995.

BANERJEE, U.; BLISS, B.; MA, Z.; PETERSEN, P. Unraveling Data Race Detection in the IntelÂ® Thread Checker. In: FIRST WORKSHOP ON SOFTWARE TOOLS FOR MULTI-CORE SYSTEMS (STMCS), IN CONJUNCTION WITH IEEE/ACM INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION (CGO), 2006. **Anais...** [S.l.: s.n.], 2006. v.26.

BARBER, C. B.; DOBKIN, D. P.; HUHDANPAA, H. The quickhull algorithm for con-

vex hulls. **ACM Trans. Math. Softw.**, New York, NY, USA, v.22, p.469–483, December 1996.

BASUPALLI, V.; YUKI, T.; RAJOPADHYE, S.; MORVAN, A.; DERRIEN, S.; QUINTON, P.; WONNACOTT, D. ompVerify: polyhedral analysis for the openmp programmer. In: CHAPMAN, B.; GROPP, W.; KUMARAN, K.; MÜLLER, M. (Ed.). **OpenMP in the Petascale Era**. [S.l.]: Springer Berlin / Heidelberg, 2011. p.37–53. (Lecture Notes in Computer Science, v.6665). 10.1007/978-3-642-21487-5_4.

BEATON, W.; RIVIERES, J. des. **Eclipse Platform Technical Overview**. Ottawa, Ontario, Canada: The Eclipse Foundation, 2006.

CHAPMAN, B.; JOST, G.; PAS, R. van der. **Using OpenMP** : portable shared memory parallel programming. Cambridge, Massachusetts: The MIT Press, 2007.

CHEN, N.; OVERBEY, J. **Photran 7.0 Developer's Guide**. [S.l.: s.n.], 2010.

DE, V. **A Foundation for Refactoring Fortran 90 in Eclipse**. 2004. Dissertação de Mestrado — University of Illinois, Urbana-Champaign, EUA.

DIG, D. A Refactoring Approach to Parallelism. **IEEE Software**, Los Alamitos, CA, USA, v.28, p.17–22, January 2011.

DIG, D.; MARRERO, J.; ERNST, M. D. Refactoring sequential Java code for concurrency via concurrent libraries. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 31., 2009, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.397–407. (ICSE '09).

DIG, D.; TARCE, M.; RADOI, C.; MINEA, M.; JOHNSON, R. Relooper: refactoring for loop parallelism in java. In: PROCEEDING OF THE 24TH ACM SIGPLAN CONFERENCE COMPANION ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, 2009, New York, NY, USA. **Anais...** ACM, 2009. p.793–794. (OOPSLA '09).

DINNING, A.; SCHONBERG, E. Detecting access anomalies in programs with critical sections. **SIGPLAN Not.**, New York, NY, USA, v.26, p.85–96, December 1991.

DORTA, A. J.; RODRÍGUEZ, C.; SANDE, F. de. **OmpSCR: openmp source code repository**. Disponível em: http://sourceforge.net/projects/ompscr/. Acesso em: outubro de 2011.

DRAGAN-CHIRILA, J. **Integrating a Fortran 90 Parser into an Eclipse Environment**. 2004. Dissertação (Mestrado) — University of Illinois, Urbana-Champaign, EUA.

ECLIPSE.ORG eclipse.org. **Photran - An Integrated Development Environment for Fortran**. Disponível em: http://www.eclipse.org/photran/. Acesso em: agosto de 2011.

eclipse.org eclipse.org eclipse.org eclipse.org. **Eclipse C/C++ Development Tooling - CDT**. Disponível em: http://www.eclipse.org/cdt/. Acesso em: agosto de 2011.

EIPE, R. M. **Extending Eclipse to create an IDE plugin for a new language with FORTRAN as a case study**. 2004. Dissertação (Mestrado) — University of Illinois, Urbana-Champaign, EUA.

ENGLER, D.; ASHCRAFT, K. RacerX: effective, static detection of race conditions and deadlocks. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.37, p.237–252, October 2003.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. **Refactoring**: improving the design of existing code. [S.l.]: Addison Wesley, 1999.

GRISWOLD, W. G.; NOTKIN, D. Automated Assistance for Program Restructuring. **ACM Transactions on Software Engineering and Methodology**, [S.l.], v.2, n.3, p.228–269, 1993.

HARROW, J. J. Runtime Checking of Multithreaded Applications with Visual Threads. In: INTERNATIONAL SPIN WORKSHOP ON SPIN MODEL CHECKING AND SOFTWARE VERIFICATION, 7., 2000, London, UK. **Proceedings...** Springer-Verlag, 2000. p.331–342.

HERMANNS, M. **Parallel Programming in Fortran 95 using OpenMP**. [S.l.]: Online, 2002.

IEROTHEOU, C. S.; JOHNSON, S. P.; CROSS, M.; LEGGETT, P. F. Computer aided parallelisation tools (CAPTools)-conceptual overview and performance on the paralleli-

sation of structured mesh codes. **Parallel Comput.**, Amsterdam, The Netherlands, The Netherlands, v.22, p.163–195, February 1996.

INRIA. **Gecos - Generic compiler suite**. Disponível em: http://gecos.gforge.inria.fr/doku.php. Acesso em: Novembro de 2011.

jchord jchord jchord jchord. **jchord - A Static and Dynamic Program Analysis Platform for Java**. Disponível em: http://code.google.com/p/jchord/. Acesso em: agosto de 2011.

JIN, H.; FRUMKIN, M.; YAN, J. Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes. In: VALERO, M.; JOE, K.; KITSUREGAWA, M.; TANAKA, H. (Ed.). **High Performance Computing**. [S.l.]: Springer Berlin / Heidelberg, 2000. p.440–456. (Lecture Notes in Computer Science, v.1940). 10.1007/3-540-39999-2_42.

JONES, J. Abstract Syntax Tree Implementation Idioms. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS (PLOP2003), 10., 2003, Allerton Park in Monticello, IL. **Proceedings...** The Hillside Group: Inc., 2003.

KERIEVSKY, J. **Refactoring to Patterns**. Boston, Massachusetts: Addison-Wesley Professional, 2004.

KIMURA, K.; WADA, Y.; NAKANO, H.; KODAKA, T.; SHIRAKO, J.; ISHIZAKA, K.; KASAHARA, H. Multigrain parallel processing on compiler cooperative chip multiprocessor. In: INTERACTION BETWEEN COMPILERS AND COMPUTER ARCHITECTURES, 2005. INTERACT-9. 9TH ANNUAL WORKSHOP ON, 2005. **Anais...** [S.l.: s.n.], 2005. p.11–20.

KOFFMANN, E.; FRIEDMAN, F. **FORTRAN**. 5.ed. [S.l.]: Addison Wesley, 2006.

LAMPORT, L. Ti clocks, and the ordering of events in a distributed system. **Commun. ACM**, New York, NY, USA, v.21, p.558–565, July 1978.

LEVESON, N. G.; TURNER, C. S. An Investigation of the Therac-25 Accidents. **IEEE Computer**, [S.l.], v.26, p.18–41, 1993.

MICROSYSTEMS, S. **Sun Studio 12**: thread analyzer user's guide. Santa Clara, CA 95054 U.S.A.: Sun Microsystems, Inc, 2007.

MITRA, S.; KOTHARI, S.; CHO, J.; KRISHNASWAMY, A. ParAgent: a domain-specific semi-automatic parallelization tool. In: VALERO, M.; PRASANNA, V.; VAJAPEYAM, S. (Ed.). **High Performance Computing - HiPC 2000**. [S.l.]: Springer Berlin / Heidelberg, 2000. p.141–148. (Lecture Notes in Computer Science, v.1970). 10.1007/3-540-44467-X_13.

NAIK, M.; AIKEN, A.; WHALEY, J. Effective static race detection for Java. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2006., 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.308–319. (PLDI '06).

NETZER, R. H. B. **Race condition detection for debugging shared-memory parallel programs**. 1991. Tese (Doutorado) — UNIVERSITY OF WISCONSIN, Madison, WI, USA. UMI Order No. GAX91-34338.

NETZER, R. H. B.; MILLER, B. P. What are race conditions?: some issues and formalizations. **ACM Lett. Program. Lang. Syst.**, New York, NY, USA, v.1, p.74–88, March 1992.

NYHOFF, L.; LEESTMA, S. **Fortran 90 for Engineers and Scientists**. [S.l.]: Prentice-Hall, 1997.

O'CALLAHAN, R.; CHOI, J.-D. Hybrid dynamic data race detection. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.167–178. (PPoPP '03).

OPDYKE, W. **Refactoring Object-Oriented Frameworks**. 1992. Tese (Doutorado) — University of Illinois, Urbana-Champaign, EUA.

OPENMP. **The OpenMP API specification for parallel programming**. Disponível em: http://openmp.org/wp/. Acesso em: Novembro de 2011.

OVERBEY, J.; JOHNSON, R. Generating Rewritable Abstract Syntax Trees. In: FIRST INTERNATIONAL CONFERENCE ON SOFTWARE LANGUAGE ENGINEERING, 2009, Toulouse, Franca. **Anais...** Springer-Verlag, 2009. p.114–133.

OVERBEY, J.; XANTHOS, S.; JOHNSON, R.; FOOTE, B. Refactorings for Fortran and High-Performance Computing. In: SECOND INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR HIGH PERFORMANCE COMPUTING SYSTEM APPLICATIONS, 2005, St. Louis, EUA. **Anais...** ACM, 2005.

PATIL, R. V.; GEORGE, B. Concurrency: tools and techniques to identify concurrency issues. **MSDN Magazine**, [S.l.], June 2008.

POZNIANSKY, E.; SCHUSTER, A. Efficient on-the-fly data race detection in multithreaded C++ programs. **SIGPLAN Not.**, New York, NY, USA, v.38, p.179–190, June 2003.

PTP. **PTP - Parallel Tools Platform**. Disponível em: http://eclipse.org/ptp/. Acesso em: Novembro de 2011.

RASMUSSEN, C. E.; SQUYRES, J. M. A Case for New MPI Fortran Bindings. In: EUROPEAN PVM/MPI USERS' GROUP MEETING, 12., 2005, Sorrento, Italia. **Anais...** [S.l.: s.n.], 2005.

RISSETTI, G. **Catálogo de Refatorações para a Evolução de Programas em Linguagem Fortran**. 2011. Dissertação (Mestrado) — Universidade Federal de Santa Maria, Santa Maria, RS.

ROBERTS, D.; BRANT, J.; JOHNSON, R. An Automated Refactoring Tool. In: INTERNATIONAL CONFERENCE ON ADVANCED SCIENCE AND TECHNOLOGY, 1996, Chicago, EUA. **Anais...** [S.l.: s.n.], 1996.

ROBERTS, D.; BRANT, J.; JOHNSON, R. E. A Refactoring Tool for Smalltalk. **Theory and Practice of Object Systems**, [S.l.], v.3, n.4, p.253–263, 1997.

RÜHL, R. Evaluation of compiler generated parallel programs on three multicomputers. In: SUPERCOMPUTING, 6., 1992, New York, NY, USA. **Proceedings...** ACM, 1992. p.15–24. (ICS '92).

SAVAGE, S.; BURROWS, M.; NELSON, G.; SOBALVARRO, P.; ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. **ACM Trans. Comput. Syst.**, New York, NY, USA, v.15, p.391–411, November 1997.

SEREBRYANY, K.; ISKHODZHANOV, T. ThreadSanitizer: data race detection in practice. In: WORKSHOP ON BINARY INSTRUMENTATION AND APPLICATIONS, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.62–71. (WBIA '09).

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Sistemas Operacionais com Java**. [S.l.]: Elsevier, 2008.

TOMITA, S. S. **Metodologia Para Paralelização de Programas Científicos**. 2004. Dissertação de Mestrado em Computação Aplicada — INPE, São José dos Campos.

TOSCANI, S. S.; OLIVEIRA, R. S. de; SILVA CARISSIMI, A. da. **Sistemas Operacionais e Programação Concorrente**. [S.l.]: Sagra-Luzzatto, 2003.

TOURWÉ, T.; MENS, T. A Survey of Software Refactoring. **IEEE Transactions on Software Engineering**, [S.l.], v.30, n.2, p.126–139, 2004.

WALKO, R. L.; AVISSAR, R. The Ocean–Land–Atmosphere Model (OLAM). Part II: formulation and tests of the nonhydrostatic dynamic core. **Mon. Wea. Rev.**, [S.l.], v.136, p.4045–4062, 2008.

WILKINSON, B.; ALLEN, M. **Parallel programming**: techniques and applications using networked workstations and parallel computers. Upper Saddle River, New Jersey: Prentice Hall, 2004.

WILSON, R. P.; FRENCH, R. S.; WILSON, C. S.; AMARASINGHE, S. P.; ANDERSON, J. M.; TJIANG, S. W. K.; LIAO, S.-W.; TSENG, C.-W.; HALL, M. W.; LAM, M. S.; HENNESSY, J. L. SUIF: an infrastructure for research on parallelizing and optimizing compilers. **SIGPLAN Not.**, New York, NY, USA, v.29, p.31–37, December 1994.

WLOKA, J.; SRIDHARAN, M.; TIP, F. Refactoring for reentrancy. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, 7., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.173–182. (ESEC/FSE '09).

YU, Y.; RODEHEFFER, T.; CHEN, W. RaceTrack: efficient detection of data race conditions via adaptive tracking. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.39, p.221–234, October 2005.