Uma extensão do Eclipse para auxiliar na refatoração de código sequencial em código paralelo com OpenMP

Dionatan K. Tietzmann[1], Andrea S. Charão[2], Júlio Cezar Beal Júnior[1,2]

[1] Universidade Regional do Noroeste do Estado do RS (Unijuí) – Ijuí, RS – Brazil
[2]Universidade Federal de Santa Maria (UFSM) – Santa Maria, RS – Brazil
dionatan.k@unijui.edu.br, andrea@inf.ufsm.br, juliobeal@gmail.com

Resumo. *Este artigo apresenta uma ferramenta, implementada como uma extensão do IDE Eclipse, que auxilia o programador na refatoração de código sequencial em código paralelo com OpenMP. A interface OpenMP é um padrão para programação paralela em arquiteturas com múltiplos processadores que compartilham memória. A ferramenta desenvolvida verifica automaticamente os acessos a dados em trechos de um programa os quais pretende-se paralelizar, apontando variáveis que podem estar envolvidas em condições de corrida. A fim de validar a ferramenta, apresenta-se um conjunto de testes e um estudo de caso baseado em um programa de grande porte, nos quais verificou-se habilidades e limitações da extensão desenvolvida.*

Abstract. *This article presents a tool, implemented as an extension of the Eclipse IDE, which helps the programmer in code refactoring sequential parallel code with OpenMP. The OpenMP is a standard interface for programming parallel architectures with multiple processors sharing memory. The developed tool automatically checks the data access in parts of a program which aims to parallelize, pointing variables that may be involved in race conditions. In order to validate the tool, presents a set of tests and a case based on a large program in which there was abilities and limitations of the developed length.*

1. Introdução

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 and 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 permitem expressar a decomposição do programa em múltiplas *threads* cooperantes. Nesse contexto, destaca-se OpenMP [openmp.org 2011], uma API padrão para programação paralela em arquiteturas com memória compartilhada, para linguagens C/C++ e Fortran. OpenMP facilita a paralelização de um programa por meio de diretivas de compilação. Assim, não é necessário gerenciar *threads* explicitamente, bastando 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 refatoração de código sequencial em código paralelo nem sempre é uma tarefa trivial. O sucesso da 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, como é o caso das condições de corrida [Silberschatz et al. 2008].

Neste trabalho, apresenta-se uma ferramenta que visa auxiliar o programador durante a refatoração de código sequencial em código paralelo com OpenMP, identificando de forma automatizada variáveis que podem vir a ter problemas de condição de corrida. Esta ferramenta é implementada como uma extensão do IDE Eclipse, utilizando o framework Photran, que oferece suporte à refatoração de código em Fortran.

O artigo está organizado como segue. A seção 2 apresenta uma contextualização sobre OpenMP, sobre refatoração de código sequencial em código paralelo e sobre o framework Photran. Ao longo desta seção também são apontados alguns trabalhos de pesquisa relacionados. A seção 3 descreve o algoritmo projetado para analisar acessos a dados. Este algoritmo é o cerne da ferramenta implementada. A seção 4 descreve a implementação da ferramenta como uma extensão do IDE Eclipse. A seção 5 apresenta os resultados da avaliação da ferramenta por meio de estudos de caso. Por fim, a seção 6 apresenta as considerações finais.

2. Fundamentação e Trabalhos Relacionados

2.1. OpenMP

O padrão OpenMP oferece uma interface simples para o desenvolvimento de aplicações paralelas, seguindo um modelo de programação baseado em *threads* que cooperam via memória compartilhada. A figura 1 apresenta um laço de repetição em linguagem Fortran, paralelizado com diretivas OpenMP. Em uma arquitetura paralela com 4 núcleos de processamento, por exemplo, a execução será realizada simultaneamente por 4 *threads*, que executarão 100 iterações do laço mais externo.

No exemplo da figura 1, é definido também quais variáveis serão privadas e quais serão compartilhadas pelas *threads* (diretivas `private` e `shared`). No caso de variáveis compartilhadas, faz-se necessária uma análise criteriosa dos acessos (leituras e escritas), para garantir que o programa paralelo 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 et al. 2008].

```
...
!$omp parallel do default(none)      &
!$omp shared(a,b,c) private(i,j)
      do i = 1, 400
         a(i) = 0.0
         do j = 1, 200
            a(i) = a(i) + b(i,j)*c(j)
         end do
      end do
!$omp end parallel do
...
```

Figura 1. Exemplo de paralelização de laço de repetição em OpenMP para Fortran

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, Serebryany and Iskhodzhanov 2009, Yu et al. 2005, Microsystems 2007, Naik et al. 2006, Engler and 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 et al. 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.

2.2. Refatoração de Código Sequencial em Código Paralelo

Em alguns trabalhos recentes [Dig et al. 2009a, Dig et al. 2009b, Wloka et al. 2009], aborda-se a transformação de um programa sequencial em um programa paralelo como um conjunto de refatorações. Entende-se por refatoração uma alteração interna feita num programa, preservando seu comportamento observável [Fowler et al. 1999, Tourwé and 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. Neste sentido, a introdução de uma diretiva OpenMP em um programa sequencial pode ser vista como uma refatoração do código. Seguindo este raciocínio, este trabalho utiliza-se dos conceitos de refatoração e análise estática de código para construir uma ferramenta que auxilie na refatoração de código serial em código paralelo.

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 oferecem infraestruturas para representação de programas em memória, que podem ser usadas como base para análise estática de código. No entanto, o uso de IDEs para a paralelização de código ainda é um tema pouco explorado, sendo esse abordado por projetos recentes, ainda em desenvolvimento [Basupalli et al. 2011]. Além disso, em geral, dentre os trabalhos que visam analisar acessos concorrentes a dados e detectar condições de corrida, são poucos [Dig et al. 2009a, Dig et al. 2009b] os que se integram a IDEs amplamente utilizados, como por exemplo o IDE Eclipse com o *plugin* Photran.

2.3. Eclipse e Photran

O Eclipse é um IDE com suporte para diversas linguagens de programação, tais como Java, C e C++. Recentemente, por meio do *plugin* Photran [eclipse.org 2011], o Eclipse passou também a suportar a linguagem Fortran, que é ativamente usada em aplicações de alto desempenho. Photran é projeto oficial da Eclipse Foundation e faz parte de um macro projeto denominado PTP (*Parallel Tools Platform*).

Um dos principais objetivos do projeto Photran é disponibilizar um *framework* que possibilite o desenvolvimento rápido de ações de refatoração para código Fortran, reutilizando a infra-estrutura fornecida pelo Eclipse [De 2004]. Para isso, são utilizadas reescritas de árvores sintáticas abstratas (ASTs), as quais são manipuladas através da adição, da modificação e da remoção de nós e informações desses nós em suas estruturas. Por ser uma ferramenta de código aberto, possuir editor de código e *parser* para a linguagem Fortran, Photran possibilitou e motivou seu uso no desenvolvimento deste trabalho.

3. Algoritmo de Análise de Acessos a Dados

Para atingir o objetivo de auxiliar na refatoração de um código sequencial para paralelo, desenvolveu-se 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.

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.

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.

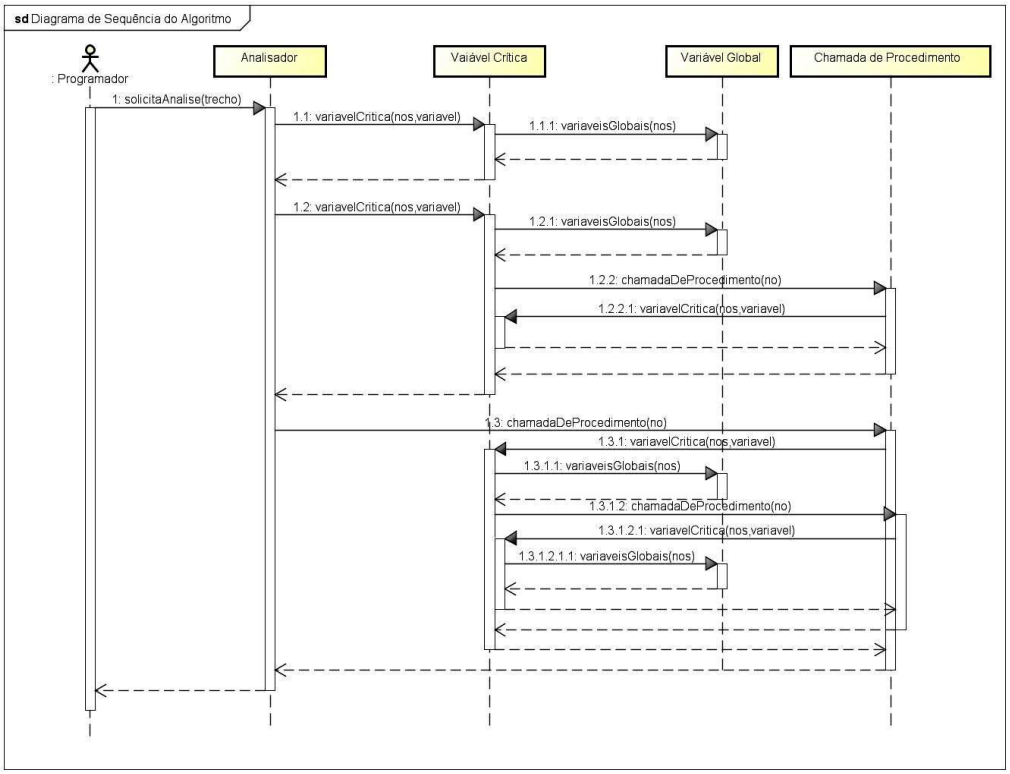

Figura 2. Diagrama de sequência do algoritmo

O algoritmo de análise divide-se em quatro partes, conforme mostra o diagrama de sequência apresentado na figura 2. O **analisador** recebe o trecho de código de entrada e, com o auxílio do analisador sintático, 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 a ser envolvida em uma condição de corrida em uma execução paralela. Caso existam variáveis globais no código de entrada, a análise de **variáveis globais** verifica se elas se enquadram em uma possível condição de corrida. Isso é necessário pois as variáveis globais podem estar sendo manipuladas em diferentes partes do código, como por exemplo, em procedimentos distintos.

A análise de **chamada de procedimento**, por sua vez, aplica a análise de variável crítica no código de procedimentos que porventura sejam invocados no código de entrada.

Isso é necessário porque um procedimento pode ler e alterar alguma variável do trecho selecionado. As interações entre as partes do algoritmo podem ser vistas na figura 2.

Neste estágio do trabalho, estruturas de dados como vetores, matrizes e tipos estruturados foram considerados como variáveis escalares. Esta simplificação leva a um menor grau de confiança na detecção, sendo que isso é indicado ao usuário. Além disso, não está 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.

4. Implementação

O algoritmo de análise 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 implementação do algoritmo, a verificação foi dividida em dois níveis a serem escolhidos pelo programador:

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

A escolha entre um destes níveis interferirá na execução do algoritmo e também nos resultados apresentados. 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 um pacote criado separadamente do Photran. Esse pacote foi desenvolvido como um novo *plugin* para o Eclipse, o qual foi chamado de ***Possible Critical Section*** e implementado em Java como "*br.ufsm.inf.viewCriticalSection*".

A figura 3 mostra um exemplo de um trecho de código analisado, tendo como um dos resultados a tabela de variáveis detectadas (parte inferior da figura). A tabela resultante da detecção possui as seguintes colunas:

- ***Image***: Identifica 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 a ferramenta ainda tem um nível baixo de precisão. Já a imagem na cor vermelha indica uma maior confiança no resultado.
- ***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.

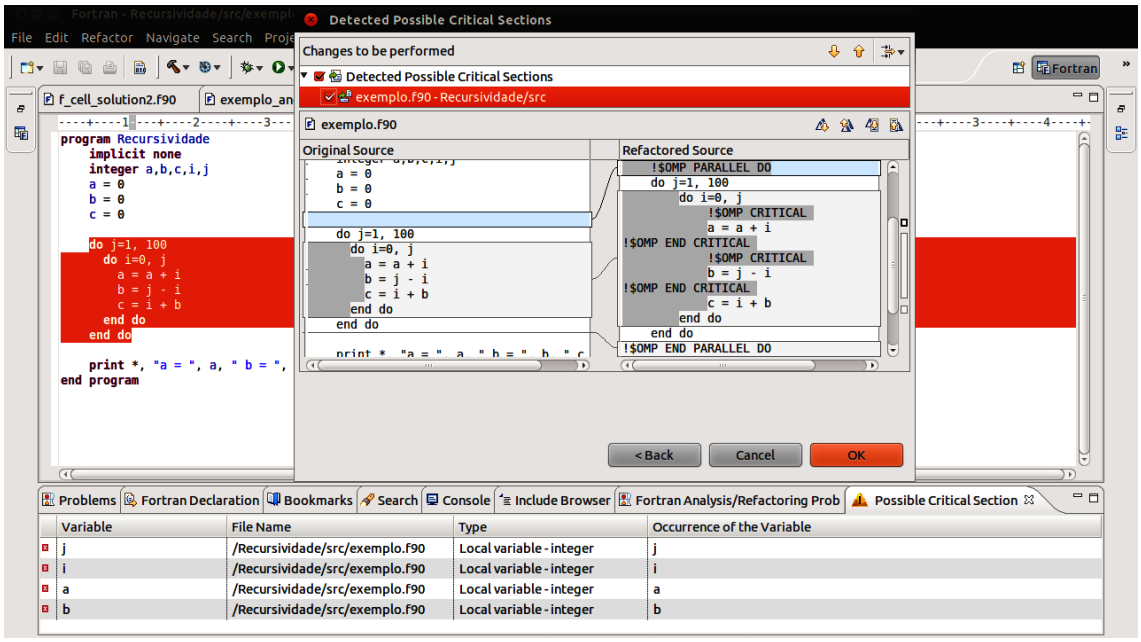

Figura 3. Exemplo de utilização da ferramenta

- ***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.

Os resultados do analisador abrem a possibilidade de inserir sugestões de instruções OpenMP para proteção das variáveis detectadas. Para explorar esta via, implementou-se um mecanismo que adiciona sugestões de instruções OpenMP no código. No atual estágio do trabalho, isso é feito 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, como pode ser visto na figura 3. As instruções implementadas são comentários adicionados no código por meio de funcionalidades de manipulação da AST providas pelo Photran. São elas:

- `!$omp parallel` e `!$omp end parallel` : delimitam o trecho de código paralelizado;
- `!$omp parallel do` e `!$omp end parallel do` : delimitam o trecho de código paralelizado caso este seja um laço de repetição “DO”;
- `private(<var>)` e `!$omp critical` : utilizados para proteger variáveis críticas;

- `!*** Warning ***`
  `!Critical variables identified in the call: <lista var>`
  `call subrotina(<lista var>)`
  : não é uma instrução OpenMP, apenas um comentário para alertar ao programador sobre variáveis possivelmente em risco na chamada da sub-rotina.

5. Avaliação

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 [Dorta et al. 2009]. 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 and Avissar 2008], uma aplicação de computação científica utilizada em produção e em pesquisa. Em ambos os casos, os testes foram concentrados em trechos de códigos já paralelizados com instruções OpenMP, para permitir uma comparação entre a paralelização manual e a paralelização com auxílio do analisador.

Os resultados apresentados pela ferramenta foram contabilizados através da análise manual dos trechos de código e das variáveis contidas nas instruções OpenMP de cada trecho. Desta forma, foram identificadas as variáveis em risco confrontando-as com o resultado obtido pela ferramenta, conhecendo-se então, o seu percentual de acerto.

5.1. Aplicações OpenMP

Os primeiros testes foram realizados com as aplicações *Pi*, *Cellular Automata*, *Jacobi* e *Molecular Dynamic*. Esses programas fazem parte do repositório OmpSCR - *OpenMP Source Code Repository* [Dorta et al. 2009], um repositório de códigos-fonte livres em OpenMP, úteis para testes.

Durante esses testes foram analisados oito trechos de código, onde a ferramenta identificou trinta e nove variáveis com possibilidade de apresentar condição de corrida. A tabela 1 apresenta com mais detalhes os dados estatísticos desses testes.

Tabela 1. Variáveis detectadas

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

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 risco de condição de corrida. As demais variáveis representam acertos da ferramenta, 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 so-

bre o programa OLAM, representa uma avaliação do analisador em um código de grande porte.

5.2. 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 and 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. Seu código é distribuído em 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 de grande porte, representativo do tipo de aplicação que pode se beneficiar do analisador desenvolvido.

```
!$omp parallel do private(iw,ka,k)
do j = 1,jtab_w(17)%jend(mrl); iw = jtab_w(17)%iw(j)
!------------------------------------------------------
   call qsub('W',iw)
   ka = lpw(iw)
   do k = ka,mza-1
      alpha_press(k,iw) = pc1                                  &
         * (((1. - sh_w(k,iw)) * rdry + sh_v(k,iw) * rvap)  &
         * theta(k,iw) / thil(k,iw)) ** cpocv
   enddo
enddo
!$omp end parallel do
```

Figura 4. Trecho de código – OLAM

A figura 4 mostra um dos trechos analisados no OLAM. 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.

Como pode ser visto na figura 5, 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`.

Os testes com o programa OLAM totalizaram 100 trechos de código que já estavam paralelizados com OpenMP. A tabela 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

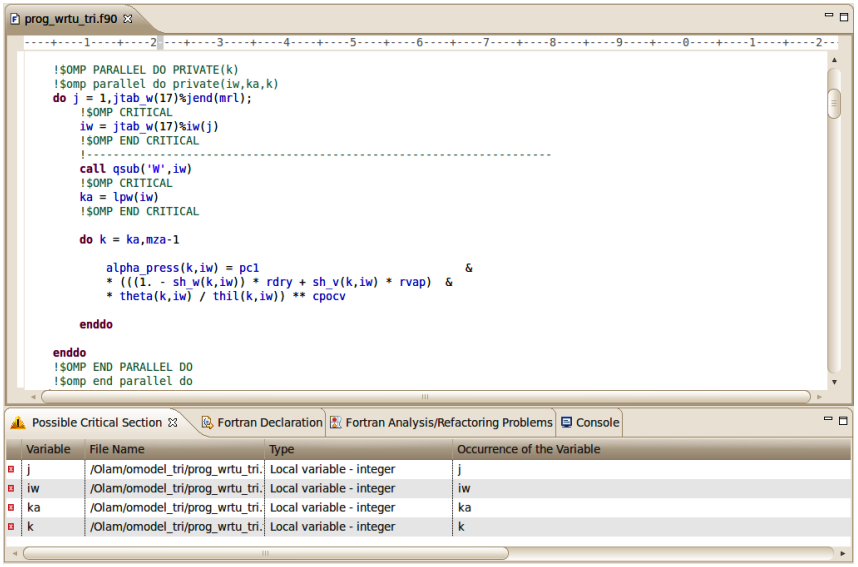

Figura 5. Resultado da análise do trecho - OLAM

nos testes anteriores, não foram identificados falsos negativos, ou seja, a ferramenta não deixou de apontar nenhuma variável em risco.

Tabela 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 às 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.

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 programador para a paralelização do trecho.

6. Consideraçoes Finais

Neste artigo, 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 paralelização, 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 de forma automatizada.

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. O código-fonte da ferramenta está disponível em: http://code.google.com/p/hpcfact/.

Como trabalhos futuros, pretende-se propor e integrar à ferramenta um algoritmo adequado para sugestão de diretivas OpenMP baseadas no resultado da análise. Além disso, serão estudadas 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

Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., and Wonnacott, D. (2011). ompverify: Polyhedral analysis for the openmp programmer. In Chapman, B., Gropp, W., Kumaran, K., and Müller, M., editors, *OpenMP in the Petascale Era*, volume 6665 of *Lecture Notes in Computer Science*, pages 37–53. Springer Berlin / Heidelberg. 10.1007/978-3-642-21487-5_4.

De, V. (2004). A foundation for refactoring fortran 90 in eclipse. Dissertação de mestrado, University of Illinois, Urbana-Champaign, EUA.

Dig, D., Marrero, J., and Ernst, M. D. (2009a). Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, Washington, DC, USA. IEEE Computer Society.

Dig, D., Tarce, M., Radoi, C., Minea, M., and Johnson, R. (2009b). Relooper: refactoring for loop parallelism in java. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 793–794, New York, NY, USA. ACM.

Dorta, A. J., Rodríguez, C., and de Sande, F. (2009). OmpSCR: OpenMP Source Code Repository. Disponível em: http://sourceforge.net/projects/ompscr/. Acesso em: outubro de 2011.

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

Engler, D. and Ashcraft, K. (2003). Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37:237–252.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

Microsystems, S. (2007). *Sun Studio 12: Thread Analyzer User's Guide*. Sun Microsystems, Inc, Santa Clara, CA 95054 U.S.A.

Naik, M., Aiken, A., and Whaley, J. (2006). Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 308–319, New York, NY, USA. ACM.

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

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

Serebryany, K. and Iskhodzhanov, T. (2009). Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA. ACM.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2008). *Sistemas Operacionais com Java*. Elsevier.

Tourwé, T. and Mens, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Walko, R. L. and Avissar, R. (2008). The ocean–land–atmosphere model (olam). part ii: Formulation and tests of the nonhydrostatic dynamic core. *Mon. Wea. Rev.*, 136:4045–4062.

Wilkinson, B. and Allen, M. (2004). *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice Hall, Upper Saddle River, New Jersey.

Wloka, J., Sridharan, M., and Tip, F. (2009). Refactoring for reentrancy. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 173–182, New York, NY, USA. ACM.

Yu, Y., Rodeheffer, T., and Chen, W. (2005). Racetrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39:221–234.